

1.1 Data structures

Data structures

A **data structure** is a way of organizing, storing, and performing operations on data. Operations performed on a data structure include accessing or updating stored data, searching for specific data, inserting new data, and removing data. The following provides a list of basic data structures.

Table 1.1.1: Basic data structures.

Data structure	Description
Record	A record is the data structure that stores subitems, often called fields, with a name associated with each subitem.
Array	An array is a data structure that stores an ordered list of items, where each item is directly accessible by a positional index.
Linked list	A linked list is a data structure that stores an ordered list of items in nodes, where each node stores data and has a pointer to the next node.
Binary tree	A binary tree is a data structure in which each node stores data and has up to two children, known as a left child and a right child.
Hash table	A hash table is a data structure that stores unordered items by mapping (or hashing) each item to a location in an array.
Heap	A max-heap is a tree that maintains the simple property that a node's key is greater than or equal to the node's childrens' keys. A min-heap is a tree that maintains the simple property that a node's key is less than or equal to the node's childrens' keys.
Graph	A graph is a data structure for representing connections among items, and consists of vertices connected by edges. A vertex represents an item in a graph. An edge represents a connection between two vertices in a graph.





- 1) A linked list stores items in an unspecified order.

True
 False

- 2) A node in binary tree can have zero, one, or two children.

True
 False

- 3) A list node's data can store a record with multiple subitems.

True
 False

- 4) Items stored in an array can be accessed using a positional index.

True
 False

©zyBooks 05/24/23 23:21 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Choosing data structures

The selection of data structures used in a program depends on both the type of data being stored and the operations the program may need to perform on that data. Choosing the best data structure often requires determining which data structure provides a good balance given expected uses. Ex: If a program requires fast insertion of new data, a linked list may be a better choice than an array.

PARTICIPATION ACTIVITY

1.1.2: A linked list avoids the shifting problem.



Animation content:

undefined

©zyBooks 05/24/23 23:21 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Animation captions:

1. Inserting an item at a specific location in an array requires making room for the item by shifting higher-indexed items.
2. Once the higher index items have been shifted, the new item can be inserted at the desired index.
3. To insert new item in a linked list, a list node for the new item is first created.

4. Item B's next pointer is assigned to point to item C. Item A's next pointer is updated to point to item B. No shifting of other items was required.

PARTICIPATION ACTIVITY

1.1.3: Basic data structures.



- 1) Inserting an item at the end of a 999-item array requires how many items to be shifted?

Check**Show answer**

©zyBooks 05/24/23 23:21 169246

Taylor Larrechea

COLORADOCSPB2270Summer2023

- 2) Inserting an item at the end of a 999-item linked list requires how many items to be shifted?

Check**Show answer**

- 3) Inserting an item at the beginning of a 999-item array requires how many items to be shifted?

Check**Show answer**

- 4) Inserting an item at the beginning of a 999-item linked list requires how many items to be shifted?

Check**Show answer**

©zyBooks 05/24/23 23:21 169246

Taylor Larrechea

COLORADOCSPB2270Summer2023

1.2 Abstract data types

Abstract data types (ADTs)

An **abstract data type (ADT)** is a data type described by predefined user operations, such as "insert data at rear," without indicating how each operation is implemented. An ADT can be implemented using different underlying data structures. However, a programmer need not have knowledge of the underlying implementation to use an ADT.

Ex: A list is a common ADT for holding ordered data, having operations like append a data item,⁴⁶² remove a data item, search whether a data item exists, and print the list. A list ADT is commonly implemented using arrays or linked list data structures.

PARTICIPATION
ACTIVITY

1.2.1: List ADT using array and linked lists data structures.



Animation captions:

1. A new list named agesList is created. Items can be appended to the list. The items are ordered.
2. Printing the list prints the items in order.
3. A list ADT is commonly implemented using array and linked list data structures. But, a programmer need not have knowledge of which data structure is used to use the list ADT.

PARTICIPATION
ACTIVITY

1.2.2: Abstract data types.



- 1) Starting with an empty list, what is the list contents after the following operations?

Append(list, 11)

Append(list, 4)

Append(list, 7)

4, 7, 11

7, 4, 11

11, 4, 7

- 2) A remove operation for a list ADT removes the specified item. Given a list with contents: 2, 20, 30, what is the list contents after the following operation?

Remove(list, item 2)

2, 30

©zyBooks 05/24/23 23:21 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



2, 20, 30 20, 30

- 3) A programmer must know the underlying implementation of the list ADT in order to use a list.

 True False

©zyBooks 05/24/23 23:21 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

- 4) A list ADT's underlying data structure has no impact on the program's execution.

 True False

Common ADTs

Table 1.2.1: Common ADTs.

Abstract data type	Description	Common underlying data structures
List	A list is an ADT for holding ordered data.	Array, linked list
Dynamic array	A dynamic array is an ADT for holding ordered data and allowing indexed access.	Array
Stack	A stack is an ADT in which items are only inserted on or removed from the top of a stack.	Linked list
Queue	A queue is an ADT in which items are inserted at the end of the queue and removed from the front of the queue.	Linked list ©zyBooks 05/24/23 23:21 1692462 Taylor Larrechea
Deque	A deque (pronounced "deck" and short for double-ended queue) is an ADT in which items can be inserted and removed at both the front and back.	Linked list COLORADOCSPB2270Summer2023
Bag	A bag is an ADT for storing items in which the order does not matter and duplicate items are allowed.	Array, linked list

Set	A set is an ADT for a collection of distinct items.	Binary search tree, hash table
Priority queue	A priority queue is a queue where each item has a priority, and items with higher priority are closer to the front of the queue than items with lower priority.	Heap
Dictionary (Map)	A dictionary is an ADT that associates (or maps) keys with values.	©zyBooks 05/24/23 23:21 1692462 Taylor Larrechea COLORADOCSPB2270Summer2023 Hash table, binary search tree

PARTICIPATION ACTIVITY

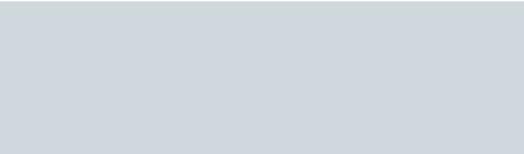
1.2.3: Common ADTs.



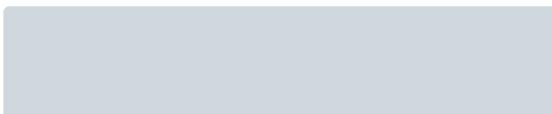
Consider the ADTs listed in the table above. Match the ADT with the description of the order and uniqueness of items in the ADT.

If unable to drag and drop, refresh the page.

Bag **Set** **Priority queue** **List**



Items are ordered based on how items are added. Duplicate items are allowed.



Items are not ordered. Duplicate items are not allowed.



Items are ordered based on items' priority. Duplicate items are allowed.



Items are not ordered. Duplicate items are allowed.

©zyBooks 05/24/23 23:21 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Reset

1.3 Applications of ADTs

Abstraction and optimization

Abstraction means to have a user interact with an item at a high-level, with lower-level internal details hidden from the user. ADTs support abstraction by hiding the underlying implementation details and providing a well-defined set of operations for using the ADT.

Using abstract data types enables programmers or algorithm designers to focus on higher-level operations and algorithms, thus improving programmer efficiency. However, knowledge of the underlying implementation is needed to analyze or improve the runtime efficiency.

PARTICIPATION ACTIVITY

1.3.1: Programming using ADTs.



Animation content:

undefined

Animation captions:

1. Abstraction simplifies programming. ADTs allow programmers to focus on choosing which ADTs best match a program's needs.
2. Both the List and Queue ADTs support efficient interfaces for removing items from one end (removing oldest entry) and adding items to the other end (adding new entries).
3. The list ADT supports iterating through list contents in reverse order, but the queue ADT does not.
4. To use the List (or Queue) ADT, the programmer does not need to know the List's underlying implementation.

PARTICIPATION ACTIVITY

1.3.2: Programming with ADTs.



Consider the example in the animation above.

- 1) The _____ ADT is the better match for the program's requirements.

- queue
- list

©zyBooks 05/24/23 23:21 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 2) The list ADT _____.

- can only be implemented using an array
- can only be implemented using a linked list



- can be implemented in numerous ways

3) Knowledge of an ADT's underlying implementation is needed to analyze the runtime efficiency.

- True
 False

©zyBooks 05/24/23 23:21 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

ADTs in standard libraries

Most programming languages provide standard libraries that implement common abstract data types. Some languages allow programmers to choose the underlying data structure used for the ADTs. Other programming languages may use a specific data structure to implement each ADT, or may automatically choose the underlying data-structure.

Table 1.3.1: Standard libraries in various programming languages.

Programming language	Library	Common supported ADTs
Python	Python standard library	list, set, dict, deque
C++	Standard template library (STL)	vector, list, deque, queue, stack, set, map
Java	Java collections framework (JCF)	Collection, Set, List, Map, Queue, Deque

PARTICIPATION ACTIVITY

1.3.3: ADTs in standard libraries.

1) Python, C++, and Java all provide built-in support for a deque ADT.

- True
 False

©zyBooks 05/24/23 23:21 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

2) The underlying data structure for a list data structure is the same for all programming languages.

True False

- 3) ADTs are only supported in standard libraries.

 True False

©zyBooks 05/24/23 23:21 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

©zyBooks 05/24/23 23:21 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

2.1 Objects: Introduction

Grouping things into objects

The physical world is made up of material items like wood, metal, plastic, fabric, etc. To keep the world understandable, people deal with higher-level objects, like chairs, tables, and TV's. Those objects are groupings of the lower-level items.

Likewise, a program is made up of items like variables and functions. To keep programs understandable, programmers often deal with higher-level groupings of those items known as objects. In programming, an **object** is a grouping of data (variables) and operations that can be performed on that data (functions).

PARTICIPATION
ACTIVITY

2.1.1: The world is viewed not as materials, but rather as objects.



Animation content:

Step 1: A list of materials from an image of a seating area reads green fabric, wood, red fabric, wood, wood, metal bar. Step 2: The material list is separated into object categories. The chair category includes the materials green fabric and wood. The couch category includes red fabric and wood. The drawer category includes wood and metal bar. Step 3: Operations are added to each of the categories. Sit is added to the chair category. Sit and lie down are added to the couch category. Put stuff in and take stuff out are added to the drawer category.

Animation captions:

1. The world consists of items like, wood, metal, fabric, etc.
2. But people think in terms of higher-level objects, like chairs, couches, and drawers.
3. In fact, people think mostly of the operations that can be done with the object. For a drawer, operations are put stuff in, or take stuff out.

PARTICIPATION
ACTIVITY

2.1.2: Programs commonly are not viewed as variables and functions/methods, but rather as objects.



Animation content:

To the left is a list of variables: Name, Phone, Cuisines, Reviews, Name, Phone, Amenities, Reviews.

To the right, these variables are split into two categories, Restaurant and Hotel.
In the restaurant category, we have the following set of operations and objects:

Set main info
Add cuisine
Add review
Print all
Name
Cuisines
Phone
Reviews

©zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

In the hotel category, we have the following set of operations and objects:

Set main info
Add amenity
Add review
Print all
Name
Amenities
Phone
Reviews

©zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Animation captions:

1. A program consists of variables and functions/methods. But programmers may prefer to think of higher-level objects like Restaurants and Hotels.
2. In fact, programmers think mostly of the operations that can be done with the object, like setting main info, or adding a review.

PARTICIPATION
ACTIVITY

2.1.3: Objects.



Some of the variables and functions for a used-car inventory program are to be grouped into an object type named CarOnLot. Select True if the item should become part of the CarOnLot object type, and False otherwise.

- 1) int carStickerPrice; □
 True
 False
- 2) double todaysTemperature; □
 True
 False
- 3) int daysOnLot; □
 True
 False
- 4) int origPurchasePrice; □
 True
 False
- 5) int numSalespeople; □
 True
 False
- 6) GetDaysOnLot() □
 True
 False
- 7) DecreaseStickerPrice() □

©zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- True
- False

8) DetermineTopSalesperson()



- True
- False

Abstraction / Information hiding

©zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Abstraction means to have a user interact with an item at a high-level, with lower-level internal details hidden from the user (aka **information hiding** or **encapsulation**). Ex: An oven supports an abstraction of a food compartment and a knob to control heat. An oven's user need not interact with internal parts of an oven.

Objects strongly support abstraction, hiding entire groups of functions and variables, exposing only certain functions to a user.

An **abstract data type (ADT)** is a data type whose creation and update are constrained to specific well-defined operations. A class can be used to implement an ADT.

PARTICIPATION ACTIVITY

2.1.4: Objects strongly support abstraction / information hiding.



Animation content:

On the left is a picture of an oven with its door closed. To the right is the oven with its door open and a woman reaching in as if to adjust the heat. Underneath this picture is a disclaimer saying "Don't do this."

Animation captions:

1. Abstraction simplifies our world. An oven is viewed as having a compartment for food, and a knob that can be turned to heat the food.
2. People need not be concerned with an oven's internal workings. Ex: People don't reach inside trying to adjust the flame.
3. Similarly, an object has operations that a user can apply. The object's internal data, and possibly other operations, are hidden from the user.

PARTICIPATION ACTIVITY

2.1.5: Abstraction / information hiding.



1) A car presents an abstraction to a user, including a steering wheel, gas pedal, and brake.



- True
- False

©zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

2) A refrigerator presents an abstraction to a user, including refrigerant gas, a compressor, and a fan.



- True
- False



3) A software object is created for a soccer team. A reasonable abstraction allows setting the team's name, adding or deleting players, and printing a roster.

- True
- False

4) A software object is created for a university class. A reasonable abstraction allows viewing and modifying variables for the teacher's name, and viewing variables for the list of students' names.

- True
- False

©zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

2.2 Using a class

Survey

The following questions are part of a zyBooks survey to help us improve our content so we can offer the best experience for students. The survey can be taken anonymously and takes just 3-5 minutes. Please take a short moment to answer by clicking the following link.

Link: [Student survey](#)

Classes intro: Public member functions

The **class** construct defines a new type that can group data and functions to form an object. A class' **public member functions** indicate all operations a class user can perform on the object. The power of classes is that a class user need not know how the class' data and functions are implemented, but need only understand how each public member function behaves. The animation below shows a class' public member function declarations only; the remainder of the class definition is discussed later.

PARTICIPATION ACTIVITY

2.2.1: A class example: Restaurant class.



©zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Animation captions:

1. A class definition creates a new type that can be used to create objects. The class declares all functions a programmer can call to operate on such an object.
2. A class user can declare a variable of the class type to create a new object.
3. Then, the class user can call the functions to operate on the object. A class user need not know how the class' data or functions are implemented.

PARTICIPATION ACTIVITY**2.2.2: Using a class.**

Consider the example above.

- 1) Which operation can a class user perform on an object of type Restaurant?

- Get the name
- Set the name
- Get the rating

©zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 2) Calling Print() on an object of type Restaurant might yield which output?

- Marias -- 5
- 5
Marias
- Marias
5

- 3) Although not visible in the part of the class definition shown above, how many internal data variables does the class contain?

- 1
- 2
- Unknown



Using a class

A programmer can create one or more objects of the same class. Declaring a variable of a class type creates an **object** of that type. Ex: `Restaurant favLunchPlace;` declares a Restaurant object named favLunchPlace.

The `".` operator, known as the **member access operator**, is used to invoke a function on an object. Ex: `favLunchPlace.SetRating(4)` calls the SetRating() function on the favLunchPlace object, which sets the object's rating to 4.

PARTICIPATION ACTIVITY**2.2.3: Using the Restaurant class.**

Animation captions:

1. Declaring a variable of the class type Restaurant creates an object of that type. The compiler allocates memory for the objects, each of which may require numerous memory locations.
2. The SetName() and SetRating() functions are invoked on the object favLunchPlace, setting that object's name to "Central Deli" and rating to 4. The object stores these values internally.
3. Invoking the SetName() and SetRating() method on the favDinnerPlace object sets that object's name to "Friends Cafe" and rating to 5.
4. Invoking the Print() operation on a Restaurant object, prints the restaurant's name and rating.

©zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

2.2.4: Using the Restaurant class.



The following questions consider *using* the Restaurant class.

- 1) Type a variable declaration that creates an object named favBreakfastPlace.

Check**Show answer**

@zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

- 2) Using separate variable declarations, create an object bestDessertPlace, followed by an object bestIndianFood.

Check**Show answer**

- 3) Given the code below, how many objects are created?

```
Restaurant bestIndianFood;
Restaurant bestSushi;
Restaurant bestCoffeeShop;
```

Check**Show answer**

- 4) Object bestSushi is of type Restaurant. Type a statement that sets the name of bestSushi to "Sushi Station".

Check**Show answer**

- 5) Type a statement to print bestCoffeeShop's name and rating.

Check**Show answer****Class example: string**@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

C++'s string type is a class. The string class stores a string's characters in memory, along with variables indicating the length and other things, but a string's user need not know such details. Instead, the string's user just needs to know what public member functions can be used, such as those shown below. (Note: size_t is an unsigned integer type).

Figure 2.2.1: Some string public member functions (many more exist).

```
char& at(size_t pos); // Returns a reference to the character at position
pos in the string.

size_t length() const; // Returns the number of characters in the string

void push_back(char c); // Appends character c to the string's end
(increasing length by 1).
```

@zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

2.2.5: Using the string class.

Consider the public member functions shown above for the string class.

- 1) Given string s = "Hi". How many bytes does object s utilize in memory?

- 2
- 3
- Unknown

- 2) Given string s = "Hi", how can a user append "!" to have s become "Hi!".

- s.push_back('!')
- s.at('!')
- Unknown

- 3) What enables a user to utilize the string class?

- Nothing; strings are built into C++
- #include <string>

2.3 Defining a class

Private data members

In addition to public member functions, a class definition has **private data members**: variables that member functions can access but class users cannot. Private data members appear after the word "private:" in a class definition.

PARTICIPATION ACTIVITY

2.3.1: Private data members.

@zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Animation captions:

1. A class definition has private data members for storing local data.
2. A class user cannot access a class' private data members; only the class' member functions can.

PARTICIPATION ACTIVITY

2.3.2: Private data members.

Consider the example above.

- 1) After declaring Restaurant x, a class user can use a statement like x.name = "Sue's Diner".

- True
- False

©zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 2) After declaring a Restaurant object x, a class user can use a statement like myString = x.name.

- True
- False

- 3) A class definition should provide comments along with each private data member so that a class user knows how those data members are used.

- True
- False

Defining a class' public member functions

A programmer defining a class first *declares* member functions after the word "public:" in the class definition. A **function declaration** provides the function's name, return type, and parameter types, but not the function's statements.

The programmer must also *define* each member function. A **function definition** provides a class name, return type, parameter names and types, and the function's statements. A member function definition has the class name and two colons (::), known as the **scope resolution operator**, preceding the function's name. A member function definition can access private data members.

PARTICIPATION ACTIVITY

2.3.3: Defining a member function of a class using the scope resolution operator.



Animation captions:

1. Without the scope resolution operator, Fct1() will yield compiler errors: numA is undefined, MyClass' Fct1()'s definition is missing.
2. Using the scope resolution operator as in MyClass::Fct1() indicates Fct1() is a member function of MyClass. Private class data becomes visible.

©zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Figure 2.3.1: A complete class definition, and use of that class.

```
#include <iostream>
#include <string>
using namespace std;

class Restaurant { // Info about
a restaurant
public:
    void SetName(string restaurantName); // Sets the
restaurant's name
    void SetRating(int userRating); // Sets the
rating (1-5, with 5 best)
    void Print(); // Prints name
and rating on one line

private:
    string name;
    int rating;
};

// Sets the restaurant's name
void Restaurant::SetName(string restaurantName) {
    name = restaurantName;
}

// Sets the rating (1-5, with 5 best)
void Restaurant::SetRating(int userRating) {
    rating = userRating;
}

// Prints name and rating on one line
void Restaurant::Print() {
    cout << name << " -- " << rating << endl;
}

int main() {
    Restaurant favLunchPlace;
    Restaurant favDinnerPlace;

    favLunchPlace.SetName("Central Deli");
    favLunchPlace.SetRating(4);

    favDinnerPlace.SetName("Friends Cafe");
    favDinnerPlace.SetRating(5);

    cout << "My favorite restaurants: " << endl;
    favLunchPlace.Print();
    favDinnerPlace.Print();

    return 0;
}
```

@zyBooks 05/27/23 21:58 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

My favorite
restaurants:
Central Deli -- 4
Friends Cafe -- 5

PARTICIPATION ACTIVITY

2.3.4: Class definition.



Consider the example above.

- 1) How is the Print() member function declared?

- Print();
- void Print();
- void Restaurant::Print();

@zyBooks 05/27/23 21:58 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023



2) How does the Print() member function's definition begin?

- void Print();
- void Restaurant::Print();
- void Restaurant::Print()

3) Could the Print() function's definition begin as follows?

```
void Restaurant::Print(int x)
```

- Yes
- No

4) Which private data members of class Restaurant do the Print() function definition's statements access?

- SetName
- favLunchPlace and favDinnerPlace
- name and rating

©zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



Example: RunnerInfo class

The RunnerInfo class below maintains information about a person who runs, allowing a class user to set the time run and the distance run, and to get the runner's speed. The subsequent question set asks for the missing parts to be completed.

Figure 2.3.2: Simple class example: RunnerInfo.

©zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

```
#include <iostream>
using namespace std;

class RunnerInfo {
public:
    void SetTime(int timeRunSecs);      // Time run in seconds
    void SetDist(double distRunMiles);   // Distance run in miles
    double GetSpeedMph() const;          // Speed in miles/hour
private:
    int timeRun;
    double distRun;
};

void ____(B)::SetTime(int timeRunSecs) {
    timeRun = timeRunSecs; // timeRun refers to data member
}

void ___(C)::SetDist(double distRunMiles) {
    distRun = distRunMiles;
}

double RunnerInfo::GetSpeedMph() const {
    return distRun / (timeRun / 3600.0); // miles / (secs / (hrs / 3600
secs))
}

int main() {
    RunnerInfo runner1; // User-created object of class type RunnerInfo
    RunnerInfo runner2; // A second object

    runner1.SetTime(360);
    runner1.SetDist(1.2);

    runner2.SetTime(200);
    runner2.SetDist(0.5);

    cout << "Runner1's speed in MPH: " << runner1.___(D) << endl;
    cout << "Runner2's speed in MPH: " << ___(E) << endl;

    return 0;
}
```

©zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Runner1's speed in MPH: 12
Runner2's speed in MPH: 9

PARTICIPATION ACTIVITY

2.3.5: Class example: RunnerInfo.



Complete the missing parts of the figure above.

1) (A)



Check

Show answer

©zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

2) (B)



Check

Show answer

3) (C)



Check**Show answer**

4) (D)

**Check****Show answer**

@zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



5) (E)

Check**Show answer**

Exploring further:

- [Classes](#) from cplusplus.com
- [Classes](#) from msdn.microsoft.com

**CHALLENGE
ACTIVITY**

2.3.1: Classes.



489394.3384924.qx3zqy7

Start

Type the program's output

```
#include <iostream>
using namespace std;

class Person {
public:
    void SetName(string nameToSet);
    string GetName() const;
private:
    string name;
};

void Person::SetName(string nameToSet) {
    name = nameToSet;
}

string Person::GetName() const {
    return name;
}

int main() {
    string userName;
    Person person1;

    userName = "Joe";

    person1.SetName(userName);
    cout << "I am " << person1.GetName();

    return 0;
}
```

I am Joe

@zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

[Check](#)[Next](#)**CHALLENGE ACTIVITY**

2.3.2: Basic class use.



Print person1's kids, apply the IncNumKids() function, and print again, outputting text as below. End each line with a newline.

Sample output for below program with input 3:

@zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Kids: 3**New baby, kids now:** 4[Learn how our autograder works](#)

489394.3384924.qx3zqy7

```

1 #include <iostream>
2 using namespace std;
3
4 class PersonInfo {
5     public:
6         void SetNumKids(int personsKidsToSet);
7         void IncNumKids();
8         int GetNumKids() const;
9     private:
10        int numKids;
11    };
12
13 void PersonInfo::SetNumKids(int personsKidsToSet) {
14     numKids = personsKidsToSet;
15 }
```

[Run](#)

View your last submission ▾

CHALLENGE ACTIVITY

2.3.3: Defining a class.



489394.3384924.qx3zqy7

[Start](#)

In the Customer class, complete the function definition for SetNumPoints() with the integer parameter

@zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Ex: If the input is 7.5 65, then the output is:

Height: 7.5**Number of points:** 65

```

1 #include <iostream>
2 using namespace std;
3
4 class Customer {
5     public:
6         void SetHeight(double customHeight);
```

```
6  
7     void SetHeight(double customHeight);  
8     double GetHeight() const;  
9     int GetNumPoints() const;  
10    private:  
11        double height;  
12        int numPoints;  
13    };  
14  
15 void Customer::SetHeight(double customHeight) {  
16 }
```

@zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

1

2

[Check](#)[Next level](#)

2.4 Inline member functions

Inline member functions

A member function's definition may appear within the class definition, known as an **inline member function**. Programmers may use inline short function definitions to yield more compact code, keeping longer function definitions outside the class definition to avoid clutter.

PARTICIPATION ACTIVITY

2.4.1: Inline member functions.



Animation content:

undefined

Animation captions:

1. A member function's definition normally appears separate from the class definition, associated with the class using the :: operator.
2. Some programmers put a short member function's definition in the class definition, for compacted code. Care must be taken not to clutter the class definition.

Figure 2.4.1: A class with two inline member functions.

@zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

```

#include <iostream>
#include <string>
using namespace std;

class Restaurant { // Info about
a restaurant
public:
    void SetName(string restaurantName) { // Sets the
restaurant's name
        name = restaurantName;
    }
    void SetRating(int userRating) { // Sets the
rating (1-5, with 5 best)
        rating = userRating;
    }
    void Print(); // Prints name
and rating on one line

private:
    string name;
    int rating;
};

// Prints name and rating on one line
void Restaurant::Print() {
    cout << name << " -- " << rating << endl;
}

int main() {
    Restaurant favLunchPlace;
    Restaurant favDinnerPlace;

    favLunchPlace.SetName("Central Deli");
    favLunchPlace.SetRating(4);

    favDinnerPlace.SetName("Friends Cafe");
    favDinnerPlace.SetRating(5);

    cout << "My favorite restaurants: " << endl;
    favLunchPlace.Print();
    favDinnerPlace.Print();

    return 0;
}

```

©zyBooks 05/27/23 21:58 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

My favorite
restaurants:
Central Deli -- 4
Friends Cafe -- 5

PARTICIPATION ACTIVITY

2.4.2: Inline member functions.



Consider the example above.

- 1) Member function SetName() was defined ____.



- inlined
- not inlined

©zyBooks 05/27/23 21:58 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

- 2) Inline member function SetRating()



____ a semicolon after the function name and parentheses, just like a function definition.

- has
- does not have



3) Member function Print() was ____.

- inlined
- not inlined

4) A function with a long definition likely
____ be inlined.



- should
- should not

5) A function defined as an inline member
function ____ also have a definition
outside the class as well.

@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



- may
- may not

Exception to variables being declared before used

Normally, items like variables must be declared before being used, but this rule does not apply within a class definition. Ex: Above, SetRating() accesses rating, even though rating is declared a few lines after. This rule exception allows a class to have the desired form of a public region at the top and a private region at the bottom: A public inline member function can thus access a private data member even though that private data member is declared after the function.

PARTICIPATION ACTIVITY

2.4.3: Inline member functions.



Consider the following class definition.

```
class PickupTruck {
public:
    void SetLength(double fullLength);
    void SetWidth (double fullWidth) {
        widthInches = fullWidth;
    }
private:
    double lengthInches;
    double widthInches;
};

void PickupTruck::SetLength(double fullLength) {
    lengthInches = fullLength;
}
```

@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

1) Inside the class definition, SetLength()
is declared but not defined.

- True
- False

2) Inside the class definition, SetWidth() is
declared but not defined.



True False

3) SetWidth() is an inline member function.

 True False

4) SetWidth()'s use of widthInches is an error because widthInches is declared after that use.

©zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

 True False

5) If the programmer defines SetWidth() inline as above, then the programmer should probably define SetLength() as inline too.

 True False

Inline member functions on one line

Normally, good style dictates putting a function's statements below the function's name and indenting. But, many programmers make an exception by putting very-short inline member function statements on the same line, for improved readability. This material may use that style at times. Example:

```
...  
void SetName(string restaurantName) { name = restaurantName; }  
void SetRating(int userRating) { rating = userRating; }  
...
```

CHALLENGE ACTIVITY

2.4.1: Inline member functions.



489394.3384924.qx3zqy7

Start

Type the program's output

©zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

```
#include <iostream>
#include <string>
using namespace std;

class Book {
public:
    void SetTitle(string bookTitle) {
        title = bookTitle;
    }
    void SetAuthor(string bookAuthor) {
        author = bookAuthor;
    }
    void Print() const;

private:
    string title;
    string author;
};

void Book::Print() const {
    cout << title << ", " << author << endl;
}

int main() {
    Book myBook;

    myBookSetTitle("The Martian");
    myBookSetAuthor("A. Weir");

    myBookPrint();

    return 0;
}
```

@zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

The Martian, A.

COLORADOCSPB2270Summer2023

1

2

[Check](#)[Next](#)

2.5 Mutators, accessors, and private helpers

Mutators and accessors

A class' public functions are commonly classified as either mutators or accessors.

- A **mutator** function may modify ("mutate") a class' data members.
- An **accessor** function accesses data members but does not modify a class' data members.

Commonly, a data member has two associated functions: a mutator for setting the value, and an accessor for getting the value, known as a **setter** and **getter** function, respectively, and typically with names starting with set or get. Other mutators and accessors may exist that aren't associated with just one data member, such as the Print() function below.

Accessor functions usually are defined as const to make clear that data members won't be changed. The keyword **const** after a member function's name and parameters causes a compiler error if the function modifies a data member. If a const member function calls another member function, that function must also be const.

@zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Figure 2.5.1: Mutator and accessor public member functions.

```
#include <iostream>
#include <string>
using namespace std;

class Restaurant {
public:
    void SetName(string restaurantName); // Mutator
    void SetRating(int userRating); // Mutator
    string GetName() const; // Accessor
    int GetRating() const; // Accessor
    void Print() const; // Accessor

private:
    string name;
    int rating;
};

void Restaurant::SetName(string restaurantName) {
    name = restaurantName;
}

void Restaurant::SetRating(int userRating) {
    rating = userRating;
}

string Restaurant::GetName() const {
    return name;
}

int Restaurant::GetRating() const {
    return rating;
}

void Restaurant::Print() const {
    cout << name << " -- " << rating << endl;
}

int main() {
    Restaurant myPlace;

    myPlace.SetName("Maria's Diner");
    myPlace.SetRating(5);

    cout << myPlace.GetName() << " is rated ";
    cout << myPlace.GetRating() << endl;

    return 0;
}
```

Maria's Diner is rated 5

@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

2.5.1: Mutators and accessors.



- 1) A mutator should not change a class' private data members.

- True
- False

- 2) An accessor should not change a class' private data members.

- True
- False

@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023





- 3) A private data member sometimes has a pair of associated set and get functions.

 True False

- 4) Accessor functions are required to be defined as const.

 True False

- 5) A const accessor function may call a non-const member function.

 True False

©zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Private helper functions

A programmer commonly creates private functions, known as **private helper functions**, to help public functions carry out tasks.

PARTICIPATION
ACTIVITY

2.5.2: Private helper member functions.



Animation captions:

1. In addition to public member functions, a class may define private member functions.
2. Any member function (public or private) may call a private member function.
3. A user of the class can call public member functions, but a user can not call private member functions (which would yield a compiler error).

PARTICIPATION
ACTIVITY

2.5.3: Private helper functions.



- 1) A class' private helper function can be called from main().

 True False

- 2) A private helper function typically helps public functions carry out their tasks.

 True False

- 3) A private helper function cannot call another private helper function.

 True False

- 4) A public member function may not call another public member function.

©zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- True
- False

CHALLENGE ACTIVITY

2.5.1: Mutators, accessors, and private helpers.



489394.3384924.qx3zqy7

Start

@zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Type the program's output

```
#include <iostream>
using namespace std;

class Dog {
public:
    void SetAge(int monthsToSet);
    string GetStage() const;
private:
    int months;
};

void Dog::SetAge(int monthsToSet) {
    months = monthsToSet;
}

string Dog::GetStage() const {
    string stage;
    if (months < 9) {
        stage = "Puppy";
    }
    else if (months < 19) {
        stage = "Adolescence";
    }
    else if (months < 50) {
        stage = "Adulthood";
    }
    else {
        stage = "Senior";
    }

    return stage;
}

int main() {
    Dog buddy;

    buddy.SetAge(1);

    cout << buddy.GetStage();
    return 0;
}
```

Puppy

1

2

3

Check**Next**

@zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

**CHALLENGE ACTIVITY**

2.5.2: Mutators, accessors, and private helpers.

489394.3384924.qx3zqy7

Start

Define the Restaurant class's SetName() mutator that sets data member name to restaurantName, followed by the SetEmployees() mutator that sets data member employees to restaurantEmployees.

Ex: If the input is Mai 29, then the output is:

Restaurant: Mai's Bakery

Total: 29 employees

```
1 #include <iostream>
2 #include <string>
3 #include <iomanip>
4 using namespace std;
5
6 class Restaurant {
7     public:
8         void SetName(string restaurantName);
9         void SetEmployees(int restaurantEmployees);
10        void Print() const;
11    private:
12        string name;
13        int employees;
14    };
15
```

©zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

1

2

3

Check

Next level

2.6 Initialization and constructors

A good practice is to initialize all variables when declared. This section deals with initializing the data members of a class when a variable of the class type is declared.

Data member initialization (C++11)

Since C++11, a programmer can initialize data members in the class definition. Any variable declared of that class type will initially have those values.

Figure 2.6.1: A class definition with initialized data members.

©zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

```
#include <iostream>
#include <string>
using namespace std;

class Restaurant {
public:
    void SetName(string restaurantName);
    void SetRating(int userRating);
    void Print();

private:
    string name = "NoName"; // NoName indicates name was not
set
    int rating = -1; // -1 indicates rating was not
set
};

void Restaurant::SetName(string restaurantName) {
    name = restaurantName;
}

void Restaurant::SetRating(int userRating) {
    rating = userRating;
}

void Restaurant::Print() {
    cout << name << " -- " << rating << endl;
}

int main() {
    Restaurant favLunchPlace; // Initializes members with
values in class definition

    favLunchPlace.Print();

    favLunchPlace.SetName("Central Deli");
    favLunchPlace.SetRating(4);

    favLunchPlace.Print();

    return 0;
}
```

@zyBooks 05/27/23 21:58 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

NoName -- -1
 Central Deli
 -- 4

PARTICIPATION
ACTIVITY

2.6.1: Initialization.



Consider the example above.

- 1) When favLunchPlace is initially declared, what is the value of favLunchPlace's rating?

Check

Show answer

@zyBooks 05/27/23 21:58 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

- 2) After the call to SetRating(), what is the value of favLunchPlace's rating?

Check

Show answer



Constructors

C++ has a special class member function, a **constructor**, called *automatically* when a variable of that class type is declared, and which can initialize data members. A constructor callable without arguments is a **default constructor**, like the Restaurant constructor below.

A constructor has the same name as the class. A constructor function has no return type, not even void. Ex:

`Restaurant::Restaurant() { . . . }` defines a constructor for the Restaurant class.

©zyBooks 05/27/23 21:58 1692462

Taylor Larrechea
COLORADOCSPB2270Summer2023

If a class has no programmer-defined constructor, then the compiler *implicitly* defines a default constructor having no statements.

Figure 2.6.2: Adding a constructor member function to the Restaurant class.

©zyBooks 05/27/23 21:58 1692462

Taylor Larrechea
COLORADOCSPB2270Summer2023

```
#include <iostream>
#include <string>
using namespace std;

class Restaurant {
public:
    Restaurant();
    void SetName(string restaurantName);
    void SetRating(int userRating);
    void Print();
private:
    string name;
    int rating;
};

Restaurant::Restaurant() { // Default constructor
    name = "NoName"; // Default name: NoName indicates name was not
set
    rating = -1; // Default rating: -1 indicates rating was not
set
}

void Restaurant::SetName(string restaurantName) {
    name = restaurantName;
}

void Restaurant::SetRating(int userRating) {
    rating = userRating;
}

// Prints name and rating on one line
void Restaurant::Print() {
    cout << name << " -- " << rating << endl;
}

int main() {
    Restaurant favLunchPlace; // Automatically calls the default constructor

    favLunchPlace.Print();

    favLunchPlace.SetName("Central Deli");
    favLunchPlace.SetRating(4);
    favLunchPlace.Print();

    return 0;
}
```

NoName -- -1
Central Deli -- 4

@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

2.6.2: Default constructors.



Assume a class named Seat.

- 1) A default constructor declaration in class Seat { ... } is:

```
class Seat {
    ...
    void Seat();
}
```

- True
- False

@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



- 2) A default constructor definition has this form:

```
Seat::Seat() {  
    ...  
}
```

- True
 False

- 3) Not defining any constructor is essentially the same as defining a constructor with no statements.

- True
 False

- 4) The following calls the default constructor once:

```
Seat mySeat;
```

- True
 False

- 5) The following calls the default constructor once:

```
Seat seat1;  
Seat seat2;
```

- True
 False

- 6) The following calls the default constructor 5 times:

```
vector<Seat> seats(5);
```

- True
 False

@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Note: Since C++11, data members can be initialized in the class definition as in `int price = -1;`, which is usually preferred over using a constructor. However, sometimes initializations are more complicated, in which case a constructor is needed.

Exploring further:

- [Constructors](#) from msdn.microsoft.com

@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

CHALLENGE
ACTIVITY

2.6.1: Enter the output of classes.

489394.3384924.qx3zqy7

Start

Type the program's output

```
#include <iostream>
#include <string>
using namespace std;

class Bicycle {
public:
    void SetType(string bicycleType);
    void SetYear(int bicycleYear);
    void Print();
private:
    string type = "NoType"; // NoType indicates brand was not set
    int year = -1; // -1 indicates year was not set
};

void Bicycle::SetType(string bicycleType) {
    type = bicycleType;
}

void Bicycle::SetYear(int bicycleYear) {
    year = bicycleYear;
}

void Bicycle::Print() {
    cout << type << " " << year << endl;
}

int main() {
    Bicycle commuterBike;

    commuterBike.Print();

    commuterBike.SetType("cross");

    commuterBike.Print();

    return 0;
}
```

@zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

NoType

CROSS -

1

2

[Check](#)[Next](#)**CHALLENGE ACTIVITY**

2.6.2: Initialization and constructors.



489394.3384924.qx3zqy7

[Start](#)

In the class definition, initialize the data members, string color, string name, and string type, with the default values "Unstated", and "Undefined", respectively.

Ex: If the input is `birch Rob mouse`, then the output is:

```
Color: Void, Name: Unstated, Type: Undefined
Color: birch, Name: Rob, Type: mouse
```

Note: The class's print function is called first after the default constructor, then again after the inputs are processed.

```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 class Animal {
6 public:
7     void SetColor(string animalColor);
8     void SetName(string animalName);
9     void SetType(string animalType);
10    void Print();
11
12 private:
```

@zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

```
13
14     /* Your code goes here */
15
```

1

2

[Check](#)[Next level](#)

©zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

2.7 Classes and vectors/classes

Vector of objects: A reviews program

Combining classes and vectors is powerful. The program below creates a `Review` class (reviews might be for a restaurant, movie, etc.), then manages a vector of `Review` objects.

Figure 2.7.1: Classes and vectors: A reviews program.

©zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

class Review {
public:
    void SetRatingAndComment(int revRating, string revComment) {
        rating = revRating;
        comment = revComment;
    }
    int GetRating() const { return rating; }
    string GetComment() const { return comment; }

private:
    int rating = -1;
    string comment = "NoComment";
};

int main() {
    vector<Review> reviewList;
    Review currReview;
    int currRating;
    string currComment;
    unsigned int i;

    cout << "Type rating + comments. To end: -1" << endl;
    cin >> currRating;
    while (currRating >= 0) {
        getline(cin, currComment); // Gets rest of line
        currReview.SetRatingAndComment(currRating,
        currComment);
        reviewList.push_back(currReview);
        cin >> currRating;
    }

    // Output all comments for given rating
    cout << endl << "Type rating. To end: -1" << endl;
    cin >> currRating;
    while (currRating != -1) {
        for (i = 0; i < reviewList.size(); ++i) {
            currReview = reviewList.at(i);
            if (currRating == currReview.GetRating()) {
                cout << currReview.GetComment() << endl;
            }
        }
        cin >> currRating;
    }

    return 0;
}
```

@zyBooks 05/27/23 21:58 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

```
Type rating +
comments. To end: -1
5 Great place!
5 Loved the food.
2 Pretty bad service.
4 New owners are nice.
2 Yuk!!!
4 What a gem.
-1

Type rating. To end:
-1
5
Great place!
Loved the food.
1
4
New owners are nice.
What a gem.
-1
```

PARTICIPATION ACTIVITY

2.7.1: Reviews program.

@zyBooks 05/27/23 21:58 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

Consider the reviews program above.

- 1) How many member functions does the Review class have?

Check

Show answer



- 2) When currReview is declared, what is the initial rating?

Check**Show answer**

- 3) As rating and comment pairs are read from input, what function adds them to vector reviewList? Type the name only, like: append.

Check**Show answer**

- 4) How many comments were output for reviews having a rating of 5?

Check**Show answer**

©zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

A class with a vector: The Reviews class

A class' private data often involves vectors. The program below redoing the example above, creating a Reviews class for managing a vector of Review objects.

The Reviews class has functions for reading reviews and printing comments. The resulting main() is clearer than above.

The Reviews class has a "getter" function returning the average rating. The function computes the average rather than reading a private data member. The class user need not know how the function is implemented.

Figure 2.7.2: Improved reviews program with a Reviews class.

©zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

```
Type ratings +  
comments. To end: -1  
5 Great place!  
5 Loved the food.  
2 Pretty bad service.  
4 New owners are nice.  
2 Yuk!!!  
4 What a gem.  
-1
```

Average rating: 3

Type rating. To end:

```
-1  
5  
5 Great place!  
Loved the food.  
1  
4  
New owners are nice.  
What a gem.  
-1
```

©zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

©zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

class Review {
public:
    void SetRatingAndComment(int revRating, string revComment) {
        rating = revRating;
        comment = revComment;
    }
    int GetRating() const { return rating; }
    string GetComment() const { return comment; }

private:
    int rating = -1;
    string comment = "NoComment";
};

// END Review class


class Reviews {
public:
    void InputReviews();
    void PrintCommentsForRating(int currRating) const;
    int GetAverageRating() const;

private:
    vector<Review> reviewList;
};

// Get rating comment pairs, add each to list. -1 rating ends.
void Reviews::InputReviews() {
    Review currReview;
    int currRating;
    string currComment;

    cin >> currRating;
    while (currRating >= 0) {
        getline(cin, currComment); // Gets rest of line
        currReview.SetRatingAndComment(currRating,
        currComment);
        reviewList.push_back(currReview);
        cin >> currRating;
    }
}

// Print all comments for reviews having the given rating
void Reviews::PrintCommentsForRating(int currRating) const {
    Review currReview;
    unsigned int i;

    for (i = 0; i < reviewList.size(); ++i) {
        currReview = reviewList.at(i);
        if (currRating == currReview.GetRating()) {
            cout << currReview.GetComment() << endl;
        }
    }
}

int Reviews::GetAverageRating() const {
    int ratingsSum;
    unsigned int i;

    ratingsSum = 0;
    for (i = 0; i < reviewList.size(); ++i) {
        ratingsSum += reviewList.at(i).GetRating();
    }
    return (ratingsSum / reviewList.size());
}
```

©zyBooks 05/27/23 21:58 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

©zyBooks 05/27/23 21:58 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY**2.7.2: Reviews program.**

Consider the reviews program above.

- 1) The first class is named Review. What is the second class named?



@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- Reviews
- reviewList
- allReviews

- 2) How many private data members does the Reviews class have?



- 0
- 1
- 2

- 3) Which function reads all reviews?



- GetReviews()
- InputReviews()

- 4) What does PrintCommentsForRating() do?



- Prints reviews sorted by rating level.
- Print all reviews above a rating level.
- Print all reviews having a particular rating level.

- 5) Does main() declare a vector?



- Yes
- No

Using Reviews in the Restaurant class

Programmers commonly use classes within classes. The program below improves the Restaurant class by having a Reviews object rather than a single rating.

@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Figure 2.7.3: Improved reviews program with a Restaurant class.

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

// Review and Reviews classes omitted from figure
// ...

class Restaurant {
public:
    void SetName(string restaurantName) {
        name = restaurantName;
    }
    void ReadAllReviews();
    void PrintCommentsByRating() const;

private:
    string name;
    Reviews reviews;
};

void Restaurant::ReadAllReviews() {
    cout << "Type ratings + comments. To end: -1"
<< endl;
    reviews.InputReviews();
}

void Restaurant::PrintCommentsByRating() const {
    int i;

    cout << "Comments for each rating level: " <<
endl;
    for (i = 1; i <= 5; ++i) {
        cout << i << ":" << endl;
        reviews.PrintCommentsForRating(i);
    }
}

int main() {
    Restaurant ourPlace;
    string currName;

    cout << "Type restaurant name: " << endl;
    getline(cin, currName);
    ourPlace.SetName(currName);
    cout << endl;

    ourPlace.ReadAllReviews();
    cout << endl;

    ourPlace.PrintCommentsByRating();

    return 0;
}
```

@zyBooks 05/27/23 21:58 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

Type restaurant name:
 Maria's Healthy Food

Type ratings + comments. To end: -1
 5 Great place!
 5 Loved the food.
 2 Pretty bad service.
 4 New owners are nice.
 2 Yuk!!!
 4 What a gem.
 -1

Comments for each rating level:
 1:
 2:
 3:
 4:
 5:
 Pretty bad service.
 Yuk!!!
 New owners are nice.
 What a gem.
 Great place!
 Loved the food.

PARTICIPATION ACTIVITY

2.7.3: Restaurant program with reviews.

@zyBooks 05/27/23 21:58 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

Consider the Restaurant program above.

- 1) How many private data members does the Restaurant class have?

- 0
- 1

2

- 2) Which Restaurant member function reads all reviews?



- GetReviews()
- InputReviews()
- ReadAllReviews()

- 3) What does PrintCommentsByRating() do?

@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- Prints comments sorted by rating level.
- Print all reviews having a particular rating level.

- 4) Does main() declare a Reviews object?



- Yes
- No

CHALLENGE ACTIVITY

2.7.1: Enter the output of classes and vectors.



489394.3384924.qx3zqy7

Start

Type the program's output

@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

class Product {
public:
    void SetPriceAndName(int productPrice, string productName) {
        price = productPrice;
        name = productName;
    };
    int GetPrice() const { return price; };
    string GetName() const { return name; };
private:
    int price; // in dollars
    string name;
};

int main() {
    vector<Product> productList;
    Product currProduct;
    int currPrice;
    string currName;
    unsigned int i;
    Product resultProduct;

    cin >> currPrice;
    while (currPrice > 0) {
        cin >> currName;
        currProduct.SetPriceAndName(currPrice, currName);
        productList.push_back(currProduct);
        cin >> currPrice;
    }

    resultProduct = productList.at(0);
    for (i = 0; i < productList.size(); ++i) {
        if (productList.at(i).GetPrice() > resultProduct.GetPrice()) {
            resultProduct = productList.at(i);
        }
    }

    cout << "$" << resultProduct.GetPrice() << " " << resultProduct.GetName() << endl;
    return 0;
}
```

@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Input
9 Tuna
11 Foil
7 Shirt
-1

Output

\$11

1

2

Check

Next

CHALLENGE ACTIVITY

2.7.2: Writing vectors with classes.



489394.3384924.qx3zqy7

Start

Write code to assign name and density properties to currMaterial, and store currMaterial in requestedMaterials. Input first receives a name value, then a density value. Input example:

Water 993 Tar 1153 quit -1

```
1 #include <iostream>
2 #include <string>
3 #include <vector>
4 using namespace std;
5
6 class Material {
7 public:
8     void SetNameAndDensity(string materialName, int materialDensity) {
9         name = materialName;
10        density = materialDensity;
11    }
12    void PrintMaterial() const {
```

@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

```

12     void PrintMaterial() const {
13         cout << name << " - " << density << endl;
14     }
15     string GetName() const { return name; }
16     int GetDensity() const { return density; }
17
18 private:
19     string name;
20     int density;
21 };
22
23 int main() {
24     vector<Material> requestedMaterials;
25     Material currMaterial;
26     string currName;
27     int currDensity;
28     unsigned int i;
29
30     cin >> currName;
31     cin >> currDensity;
32     while ((currName != "quit") && (currDensity > 0)) {
33
34     /* Your code goes here */
35

```

©zyBooks 05/27/23 21:58 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

1

2

3

Check**Next**
CHALLENGE ACTIVITY
2.7.3: Classes and vectors/classes.


489394.3384924.qx3zqy7

Start

The program first reads integer `orderCount` from input, representing the number of pairs of inputs to be read. Then it reads `orderCount` pairs of inputs, each consisting of a string and a character, representing the order's food and option, respectively. One `Order` object is created for each pair and added to vector `orderList`. If an `Order` object's option status is equal to 'D', call the `Order` object's `Print()` function.

Ex: If the input is:

```
4
egg D quail D truffle C persimmon C
```

then the output is:

```
Order: egg, Option: D
Order: quail, Option: D
```

```

1 #include <iostream>
2 #include <vector>
3 using namespace std;
4
5 class Order {
6 public:
7     void SetFoodAndOption(string newFood, char newOption);
8     char GetOption() const;
9     void Print() const;
10 private:
11     string food;
12     char option;

```

©zyBooks 05/27/23 21:58 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

```

13 };
```

```

14
15 void Order::SetFoodAndOption(string newFood, char newOption) {
16 }
```

1

2

3

Check**Next level**

©zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

2.8 Separate files for classes

Two files per class

Programmers typically put all code for a class into two files, separate from other code.

- **ClassName.h** contains the class definition, including data members and member function declarations.
- **ClassName.cpp** contains member function definitions.

A file that uses the class, such as a main file or ClassName.cpp, must include ClassName.h. The .h file's contents are sufficient to allow compilation, as long as the corresponding .cpp file is eventually compiled into the program too.

The figure below shows how all the .cpp files might be listed when compiled into one program. Note that the .h file is not listed in the compilation command, due to being included by the appropriate .cpp files.

Figure 2.8.1: Using two separate files for a class.

StoreItem.h

```

#ifndef STOREITEM_H
#define STOREITEM_H

class StoreItem {
public:
    void SetWeightOunces(int ounces);
    void Print() const;
private:
    int weightOunces;
};

#endif
```

StoreItem.cpp

```

#include <iostream>
using namespace std;

#include "StoreItem.h"

void StoreItem::SetWeightOunces(int ounces) {
    weightOunces = ounces;
}

void StoreItem::Print() const {
    cout << "Weight (ounces): " <<
    weightOunces << endl;
}
```

©zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

```
main.cpp
#include <iostream>
using namespace std;

#include "StoreItem.h"

int main() {
    StoreItem item1;

    item1.SetWeightOunces(16);
    item1.Print();

    return 0;
}
```

Compilation example

```
% g++ -Wall StoreItem.cpp main.cpp
% a.out
Weight (ounces): 16
```

©zyBooks 05/27/23 21:58 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

Good practice for .cpp and .h files

Sometimes multiple small related classes are grouped into a single file to avoid a proliferation of files. But for typical classes, good practice is to create a unique .cpp and .h file for each class.

PARTICIPATION ACTIVITY

2.8.1: Separate files.



- 1) Commonly a class definition and associated function definitions are placed in a .h file.

- True
 False

- 2) The .cpp file for a class should #include the associated .h file.

- True
 False

- 3) A drawback of the separate file approach is longer compilation times.

- True
 False



©zyBooks 05/27/23 21:58 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

Ex: Restaurant review classes

The restaurant review program, introduced in an earlier section, declared the Review, Reviews, and Restaurant classes in main.cpp. Each of the 3 classes should instead be implemented in .h/.cpp files, thus making for cleaner code in main.cpp.

Figure 2.8.2: .h and .cpp files for Review, Reviews, and Restaurant classes.

Review.h

```
#ifndef REVIEW_H
#define REVIEW_H

#include <string>

class Review {
public:
    void SetRatingAndComment(
        int revRating,
        std::string revComment);
    int GetRating() const;
    std::string GetComment() const;

private:
    int rating = -1;
    std::string comment = "NoComment";
};

#endif
```

Review.cpp

```
#include "Review.h"
using namespace std;

void Review::SetRatingAndComment(int
revRating, string revComment) {
    rating = revRating;
    comment = revComment;
}

int Review::GetRating() const {
    return rating;
}

string Review::GetComment() const {
    return comment;
}
```

@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Reviews.h

```
#ifndef REVIEWS_H
#define REVIEWS_H

#include <vector>
#include "Review.h"

class Reviews {
public:
    void InputReviews();
    void PrintCommentsForRating(int currRating) const;
    int GetAverageRating() const;

private:
    std::vector<Review> reviewList;
};

#endif
```

Reviews.cpp

```
#include <iostream>
#include "Reviews.h"
using namespace std;

// Get rating comment pairs, add each to list. -1 rating ends.
void Reviews::InputReviews() {
    Review currReview;
    int currRating;
    string currComment;

    cin >> currRating;
    while (currRating >= 0) {
        getline(cin, currComment); // Gets rest of line

        currReview.SetRatingAndComment(currRating,
                                        currComment);
        reviewList.push_back(currReview);
        cin >> currRating;
    }
}

// Print all comments for reviews having the given rating
void Reviews::PrintCommentsForRating(int currRating) const {
    Review currReview;
    unsigned int i;

    for (i = 0; i < reviewList.size(); ++i)
    {
        currReview = reviewList.at(i);
        if (currRating ==
currReview.GetRating()) {
            cout << currReview.GetComment() <<
endl;
        }
    }
}

int Reviews::GetAverageRating() const {
    int ratingsSum;
    unsigned int i;

    ratingsSum = 0;
    for (i = 0; i < reviewList.size(); ++i)
    {
        ratingsSum +=
reviewList.at(i).GetRating();
    }
    return (ratingsSum / reviewList.size());
}
```

@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Restaurant.h

```
#ifndef RESTAURANT_H
#define RESTAURANT_H

#include <string>
#include "Reviews.h"

class Restaurant {
public:
    void SetName(std::string restaurantName);
    void ReadAllReviews();
    void PrintCommentsByRating();
const;

private:
    std::string name;
    Reviews reviews;
};

#endif
```

Restaurant.cpp

```
#include <iostream>
#include "Restaurant.h"
using namespace std;

void Restaurant::SetName(string restaurantName) {
    name = restaurantName;
}

void Restaurant::ReadAllReviews() {
    cout << "Type ratings + comments. To end: -1" << endl;
    reviews.InputReviews();
}

void Restaurant::PrintCommentsByRating()
const {
    int i;

    cout << "Comments for each rating level:" << endl;
    for (i = 1; i <= 5; ++i) {
        cout << i << ":" << endl;
        reviews.PrintCommentsForRating(i);
    }
}
```

©zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

2.8.2: Restaurant reviews program's main.cpp.

**Animation content:**

undefined

Animation captions:

1. The Review, Reviews, and Restaurant classes are included in main.cpp by including Restaurant.h.
2. main()'s code is reasonably short, since reusable code resides in external files.

PARTICIPATION ACTIVITY

2.8.3: Restaurant review program .h and .cpp files.



If unable to drag and drop, refresh the page.

Reviews.cpp**Review.h****Reviews.h****Restaurant.h****Restaurant.cpp****Review.cpp**

©zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

#includes the "Restaurant.h" header file.

Uses cin and getline() statements to get ratings and comments from the user.

Makes the Restaurant, Reviews, and Review classes available when being #included by another code file.

Does not #include any of the 3 header files.

#includes the <vector> header file.

Implements class member functions, none of which use cin or cout.

@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Reset

CHALLENGE ACTIVITY

2.8.1: Enter the output of separate files.



489394.3384924.qx3zqy7

Start

Type the program's output

main.cpp **Product.h** **Product.cpp**

```
#include <iostream>
#include <vector>
#include "Product.h"
using namespace std;

int main() {
    vector<Product> productList;
    Product currProduct;
    int currPrice;
    string currName;
    unsigned int i;
    Product resultProduct;

    cin >> currPrice;
    while (currPrice > 0) {
        cin >> currName;
        currProduct.SetPriceAndName(currPrice, currName);
        productList.push_back(currProduct);
        cin >> currPrice;
    }

    resultProduct = productList.at(0);
    for (i = 0; i < productList.size(); ++i) {
        if (productList.at(i).GetPrice() < resultProduct.GetPrice()) {
            resultProduct = productList.at(i);
        }
    }

    cout << resultProduct.GetName() << ":" << resultProduct.GetPrice() << endl;
    return 0;
}
```

Input

9 Berries
8 Paper
7 Shirt
-1

Output

Shirt

@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

1

2

Check

Next

2.9 Choosing classes to create

Decomposing into classes

Creating a program may start by a programmer deciding what "things" exist, and what each thing contains and does.

Below, the programmer wants to maintain a soccer team. The programmer realizes the team will have people, so decides to sketch a Person class. Each Person class will have private (shown by "-") data like name and age, and public (shown by "+") functions like get/set name, get/set age, and print. The programmer then sketches a Team class, which uses Person objects.

PARTICIPATION
ACTIVITY

2.9.1: Creating a program by first sketching classes.



Animation content:

Programmer decides what "things" exist:

My program

Will have a soccer team

The team will have a head coach, assistant coach, list of players, name, etc.

Each coach and player will have a name, age, phone, etc.

Programmer sketches a Person class:

I need a class for a "person" (coaches, players)

Person

-name : string

-age : int

+get/set name

+get/set age

+print

Programmer sketches a Team class:

And for a "team"

Team

-head coach : Person

-asst coach : Person

+get/set head coach

+get/set asst coach

+print

More to come (list of players, name, etc.)

Animation captions:

1. A programmer thinks of what "things" a program may involve. The programmer decides one thing is a Team, and another thing is a Person.
2. The programmer sketches a Person class. Private items (shown by "-") are name and age. Public items (shown by "+") are getters/setters and print.
3. The programmer then sketches a Team class. Private items are head coach and asst coach, both of Person type. Public items are getters/setters and print.

©zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION
ACTIVITY

2.9.2: Decomposing a program into classes.



Consider the example above.

- 1) Only one way exists to decompose a program into classes.

True
 False



- 2) The - indicates a class' private item.

True
 False



@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 3) The + indicates additional private items.

True
 False



- 4) The Team class uses the Person class.

True
 False



- 5) The Person class uses the Team class.

True
 False



Coding the classes

A programmer can convert the class sketches above into code. The programmer likely would first create and test the Person class, followed by the Team class.

Figure 2.9.1: SoccerTeam and TeamPerson classes.

@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

TeamPerson.h

```
#ifndef TEAMPERSON_H
#define TEAMPERSON_H

#include <string>
using namespace std;

class TeamPerson {
public:
    void SetFullName(string firstAndLastName);
    void SetAgeYears(int ageInYears);
    string GetFullName() const;
    int GetAgeYears() const;
    void Print() const;

private:
    string fullName;
    int ageYears;
};

#endif
```

TeamPerson.cpp

```
#include <iostream>
#include <string>
using namespace std;

#include "TeamPerson.h"

void TeamPerson::SetFullName(string firstAndLastName) {
    fullName = firstAndLastName;
}

void TeamPerson::SetAgeYears(int ageInYears) {
    ageYears = ageInYears;
}

string TeamPerson::GetFullName() const {
    return fullName;
}

int TeamPerson::GetAgeYears() const {
    return ageYears;
}

void TeamPerson::Print() const {
    cout << "Full name: " << fullName
        << endl;
    cout << "Age (years): " << ageYears
        << endl;
}
```

SoccerTeam.h

```
#ifndef SOCCERTEAM_H
#define SOCCERTEAM_H

#include "TeamPerson.h"

class SoccerTeam {
public:
    void SetHeadCoach(TeamPerson teamPerson);
    void SetAssistantCoach (TeamPerson teamPerson);

    TeamPerson GetHeadCoach() const;
    TeamPerson GetAssistantCoach() const;

    void Print() const;

private:
    TeamPerson headCoach;
    TeamPerson assistantCoach;
    // Players omitted for brevity
};

#endif
```

SoccerTeam.cpp

```
#include <iostream>
using namespace std;

#include "SoccerTeam.h"

void SoccerTeam::SetHeadCoach(TeamPerson teamPerson) {
    headCoach = teamPerson;
}

void SoccerTeam::SetAssistantCoach(TeamPerson teamPerson) {
    assistantCoach = teamPerson;
}

TeamPerson SoccerTeam::GetHeadCoach
const {
    return headCoach;
}

TeamPerson SoccerTeam::GetAssistantCoach()
const {
    return assistantCoach;
}

void SoccerTeam::Print() const {
    cout << "HEAD COACH: " << endl;
    headCoach.Print();
    cout << endl;

    cout << "ASSISTANT COACH: " << endl;
    assistantCoach.Print();
    cout << endl;
}
```

@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

main.cpp

```
#include <iostream>
using namespace std;

#include "SoccerTeam.h"
#include "TeamPerson.h"

int main() {
    SoccerTeam teamCalifornia;
    TeamPerson headCoach;
    TeamPerson asstCoach;

    headCoach.SetFullName("Mark Miwerds");
    headCoach.SetAgeYears(42);
    teamCalifornia.SetHeadCoach(headCoach);

    asstCoach.SetFullName("Stanley Lee");
    asstCoach.SetAgeYears(30);

    teamCalifornia.SetAssistantCoach(asstCoach);

    teamCalifornia.Print();

    return 0;
}
```

HEAD COACH:
Full name: Mark Miwerds
Age (years): 42

ASSISTANT COACH:
Full name: Stanley Lee
Age (years): 30

@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



Consider the example above.

- 1) The programmer first sketched the desired classes, before writing the code seen above.

- True
- False

- 2) The programmer wrote one large file containing all the classes.

- True
- False

- 3) Good practice would be to first write the TeamPerson class and then test that class, followed by writing the SoccerTeam class and testing that class.

- True
- False

@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Included files

Above, note that each file only includes needed header files. SoccerTeam.h has a TeamPerson member so includes TeamPerson.h. SoccerTeam.cpp includes SoccerTeam.h. main.cpp declares objects of both types so also includes both .h files. A *common error is to include unnecessary .h files, which misleads the reader.*

Note that only .h files are included, never .cpp files.

PARTICIPATION
ACTIVITY

2.9.4: Classes and includes.



Consider the earlier SoccerTeam and TeamPerson classes. Indicate which .h files should be included in each file.

- 1) TeamPerson.h

- TeamPerson.h
- SoccerTeam.h
- No .h file needed

- 2) TeamPerson.cpp

- TeamPerson.h
- SoccerTeam.h
- No .h file needed

- 3) SoccerTeam.h

- TeamPerson.h
- SoccerTeam.h
- No .h file needed

- 4) SoccerTeam.cpp

@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- TeamPerson.h
- SoccerTeam.h
- TeamPerson.cpp
- TeamPerson.h and SoccerTeam.h

5) main.cpp

- main.h
- TeamPerson.h
- TeamPerson.h and SoccerTeam.h
- TeamPerson.cpp
- SoccerTeam.cpp



©zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

2.10 Unit testing (classes)

Testbenches

Like a chef who tastes food before serving, a class creator should test a class before allowing use. A **testbench** is a program whose job is to thoroughly test another program (or portion) via a series of input/output checks known as **test cases**. **Unit testing** means to create and run a testbench for a specific item (or "unit") like a function or a class.



PARTICIPATION
ACTIVITY

2.10.1: Unit testing of a class.



Animation content:

Three different programs are shown, each outlined with a box.

The first box contains the following:

SampleClass
Public item1
Public item2
Public item3

©zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

The second box contains the following:

User program
Create SampleClass object
Use public item 2

The third box contains the following:

SampleClassTester program
Create SampleClass object
Test public item1
Test public item2
Test public item3

Animation captions:

1. A typical program may not thoroughly use all class items.
2. A testbench's job is to thoroughly test all public class items.
3. After testing, class is ready for use. The tester program is kept for later tests.

©zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

The testbench below creates an object, then checks public functions for correctness. Some tests failed.

Features of a good testbench include:

- Automatic checks. Ex: Values are compared, as in `testData.GetNum1() != 100`. For conciseness, only fails are printed.
- Independent test cases. Ex: The test case for `GetAverage()` assigns new values, vs. relying on earlier values.
- **100% code coverage**: Every line of code is executed. A good testbench would have more test cases than below.
- Includes not just typical values but also **border cases**: Unusual or extreme test case values like 0, negative numbers, or large numbers.

Figure 2.10.1: Unit testing of a class.

©zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

```
#include <iostream>
using namespace std;

// Note: This class intentionally has errors

class StatsInfo {
public:
    void SetNum1(int numVal) { num1 = numVal; }
    void SetNum2(int numVal) { num2 = numVal; }
    int GetNum1() const { return num1; }
    int GetNum2() const { return num2; }
    int GetAverage() const;

private:
    int num1;
    int num2;
};

int StatsInfo::GetAverage() const {
    return num1 + num2 / 2;
}
// END StatsInfo class

// TESTBENCH main() for StatsInfo class
int main() {
    StatsInfo testData;

    // Typical testbench tests more thoroughly
    cout << "Beginning tests." << endl;

    // Check set/get num1
    testData.SetNum1(100);
    if (testData.GetNum1() != 100) {
        cout << "    FAILED set/get num1" << endl;
    }

    // Check set/get num2
    testData.SetNum2(50);
    if (testData.GetNum2() != 50) {
        cout << "    FAILED set/get num2" << endl;
    }

    // Check GetAverage()
    testData.SetNum1(10);
    testData.SetNum2(20);
    if (testData.GetAverage() != 15) {
        cout << "    FAILED GetAverage for 10, 20" <<
endl;
    }

    testData.SetNum1(-10);
    testData.SetNum2(0);
    if (testData.GetAverage() != -5) {
        cout << "    FAILED GetAverage for -10, 0" <<
endl;
    }

    cout << "Tests complete." << endl;
}

return 0;
}
```

©zyBooks 05/27/23 21:58 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

Beginning tests.
 FAILED set/get num2
 FAILED GetAverage for
 10, 20
 FAILED GetAverage for
 -10, 0
 Tests complete.

©zyBooks 05/27/23 21:58 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

Defining a testbench as a friend class (discussed elsewhere) enables direct testing of private member functions

2.10.2: Unit testing of a class.



- 1) A class should be tested individually (as a "unit") before use in another program.

 True False

- 2) Calling every function at least once is a prerequisite for 100% code coverage.

 True False

- 3) If a testbench achieves 100% code coverage and all tests passed, the class must be bug free.

 True False

- 4) A testbench should test all possible values, to ensure correctness.

 True False

- 5) A testbench should print a message for each test case that passes and for each that fails.

 True False

@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Regression testing

Regression testing means to retest an item like a class anytime that item is changed; if previously-passed test cases fail, the item has "regressed".

A testbench should be maintained along with the item, to always be usable for regression testing. A testbench may be in a class' file, or in a separate file as in MyClassTest.cpp for a class in MyClass.cpp.

Testbenches may be complex, with thousands of test cases. Various tools support testing, and companies employ test engineers who only test other programmers' items. A large percent, like 50% or more, of commercial software development time may go into testing.



@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 1) Testbenches are typically disposed of after use.

 True False

- 2) Regression testing means to check if a change to an item caused previously-passed test cases to fail.



True False

- 3) For commercial software, testing consumes a large percentage of time.

 True False

©zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Erroneous unit tests

An erroneous unit test may fail even if the code being tested is correct. A common error is for a programmer to assume that a failing unit test means that the code being tested has a bug. Such an assumption may lead the programmer to spend time trying to "fix" code that is already correct. Good practice is to inspect the code of a failing unit test before making changes to the code being tested.

Figure 2.10.2: Correct implementation of StatsInfo class.

```
#include <iostream>
using namespace std;

class StatsInfo {
public:
    void SetNum1(int numVal) { num1 = numVal; }
    void SetNum2(int numVal) { num2 = numVal; }
    int GetNum1() const { return num1; }
    int GetNum2() const { return num2; }
    int GetAverage() const;

private:
    int num1;
    int num2;
};

int StatsInfo::GetAverage() const {
    return (num1 + num2) / 2;
}
```

PARTICIPATION ACTIVITY

2.10.4: Erroneous unit test code causes failures even when StatsInfo is correctly implemented.



Animation content:

undefined

©zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Animation captions:

1. testData is instantiated and num1 and num2 are properly set to 20 and 30.
2. Whether a typo or miscalculation, the unit test expects 35 instead of 25, and fails. A wrong expected value is one reason a unit test may fail.
3. Calling SetNum1 twice and not calling SetNum2 is also an error, even if the expected value is now correct.
4. Not properly initializing the test object's data is another common error.

PARTICIPATION ACTIVITY

2.10.5: Identifying erroneous test cases.



Assume that StatsInfo is correctly implemented and identify each test case as valid or erroneous.

- 1) num1 = 1.5, num2 = 3.5, and the expected average = 2.5



@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- Valid
 Erroneous

- 2) num1 = 33, num2 = 11, and the expected average = 22



- Valid
 Erroneous

- 3) num1 = 101, num2 = 202, and the expected average = 152



- Valid
 Erroneous

Exploring further:

- [C++ Unit testing frameworks](#) from accu.org.

CHALLENGE ACTIVITY

2.10.1: Enter the output of the unit tests.



Note: There's always an error.

489394.3384924.qx3zqy7

Start

Type the program's output

@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

```
#include <iostream>
using namespace std;

class Rectangle {
public:
    void SetSize(int heightVal, int widthVal) {
        height = heightVal;
        width = widthVal;
    }
    int GetArea() const;
    int GetPerimeter() const;

private:
    int height;
    int width;
};

int Rectangle::GetArea() const {
    return height * width;
}

int Rectangle::GetPerimeter() const {
    return (height + width) * 2;
}

int main() {
    Rectangle myRectangle;

    myRectangle.SetSize(1, 1);
    if (myRectangle.GetArea() != 1) {
        cout << "FAILED GetArea() for 1, 1" << endl;
    }
    if (myRectangle.GetPerimeter() != 4) {
        cout << "FAILED GetPerimeter() for 1, 1" << endl;
    }

    myRectangle.SetSize(2, 3);
    if (myRectangle.GetArea() != 6) {
        cout << "FAILED GetArea() for 2, 3" << endl;
    }
    if (myRectangle.GetPerimeter() != 10) {
        cout << "FAILED GetPerimeter() for 2, 3" << endl;
    }

    return 0;
}
```

@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

FAILED GetPerimeter()

1

2

Check

Next

CHALLENGE ACTIVITY

2.10.2: Unit testing of a class.



Write a unit test for addInventory(), which has an error. Call redSweater.addInventory() with argument sweaterShipment. Print the shown error if the subsequent quantity is incorrect. Sample output for failed unit test given initial quantity is 10 and sweaterShipment is 5:

Beginning tests.

UNIT TEST FAILED: addInventory()
Tests complete.

Note: UNIT TEST FAILED is preceded by 3 spaces.

@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

[Learn how our autograder works](#)

489394.3384924.qx3zqy7

1 #include <iostream>
2 using namespace std;
3

```

4 class InventoryTag {
5 public:
6     InventoryTag();
7     int getQuantityRemaining() const;
8     void addInventory(int numItems);
9
10 private:
11     int quantityRemaining;
12 };
13
14 InventoryTag::InventoryTag() {
15     quantityRemaining = 0;
16 }

```

@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Run

View your last submission ▾

2.11 Constructor overloading

Basics

Programmers often want to provide different initialization values when creating a new object. A class creator can **overload** a constructor by defining multiple constructors differing in parameter types. A constructor declaration can have arguments. The constructor with matching parameters will be called.

PARTICIPATION
ACTIVITY

2.11.1: Overloaded constructors.



Animation content:

Shows code having two constructors, one with no parameters, and one with two parameters. Then in main(), one declaration has no arguments, and another has two arguments.

Animation captions:

1. A declaration with no arguments calls the default constructor. In this case, the object gets initialized with NoName and -1.
2. This declaration's string and int arguments match another constructor, which is called instead. The object gets initialized with those argument values.

zyDE 2.11.1: Overloading a constructor.

@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Load default template...**Run**

```

1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 class Restaurant {
6     public:
7         Restaurant();
8         Restaurant(string initName, i

```

```

9     void Print();
10
11     private:
12         string name;
13         int rating;
14     };
15

```

PARTICIPATION ACTIVITY

2.11.2: Overloaded constructors.

@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Given the three constructors below, indicate which will be called for each declaration.

```

class SomeClass {
    SomeClass();           // A
    SomeClass(string name); // B
    SomeClass(string name, int num); // C
}

```

1) `SomeClass myObj("Lee");`

- A
- B
- C
- Error



2) `SomeClass myObj();`

- A
- B
- Error



3) `SomeClass myObj;`

- A
- B
- Error



4) `SomeClass myObj("Lee", 5, 0);`

- C
- Error



5) `vector<SomeClass> myVect(5);`

- A
- Error



@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

If any constructor defined, should define default

If a programmer defines any constructor, the compiler does not implicitly define a default constructor, so good practice is for the programmer to also explicitly define a default constructor so that a declaration like `MyClass x;` remains supported.

Figure 2.11.1: Error - The programmer defined a constructor, so the compiler does not automatically define a default constructor.

```
class Restaurant {
public:
    Restaurant(string initName,
    int initRating);

    // No other constructors
    ...
};

int main() {
    Restaurant foodPlace;
    ...
}
```

tmp1.cpp:37:15: error: no matching
constructor for initialization of
'Restaurant'
 Restaurant foodPlace;

@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

2.11.3: Constructor definitions.



Which of the following is OK as the entire set of constructors for class MyClass? Assume a declaration like `MyClass x;` should be supported.

1) `MyClass();`

- OK
- Error



2) `// None`

- OK
- Error



3) `MyClass();`
`MyClass(string name);`

- OK
- Error



4) `MyClass(string name);`

- OK
- Error



Constructors with default parameter values

@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Like any function, a constructor's parameters may be assigned default values.

If those default values allow the constructor to be called without arguments, then that constructor can serve as the default constructor.

The default values could be in the function definition, but are clearer to class users in the declaration.

Figure 2.11.2: A constructor with default parameter values can serve as the default constructor.

```
#include <iostream>
#include <string>
using namespace std;

class Restaurant {
public:
    Restaurant(string initName = "NoName", int initRating =
-1);
    void Print();

private:
    string name;
    int rating;
};

Restaurant::Restaurant(string initName, int initRating) {
    name = initName;
    rating = initRating;
}

// Prints name and rating on one line
void Restaurant::Print() {
    cout << name << " -- " << rating << endl;
}

int main() {
    Restaurant foodPlace;
    Restaurant coffeePlace("Joes", 5);

    foodPlace.Print();
    coffeePlace.Print();

    return 0;
}
```

@zyBooks 05/27/23 21:58 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

NoName --
 -1
 Joes -- 5

PARTICIPATION ACTIVITY

2.11.4: Constructor with default parameter values may serve as default constructor.



Which of the following is OK as the entire set of constructors for class YourClass? Assume a declaration like `YourClass obj;` should be supported.



1) `YourClass();`

- OK
- Error



2) `YourClass(string name, int num);`

- OK
- Error

3) `YourClass(string name = "", int num = 0);`

- OK
- Error

@zyBooks 05/27/23 21:58 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023



4) `YourClass();`
`YourClass(string name = "", int num = 0);`

- OK




CHALLENGE ACTIVITY

2.11.1: Enter the output of the constructor overloading.



489394.3384924.qx3zqy7

Start

Type the program's output

```
#include <iostream>
#include <string>
using namespace std;

class Pet {
public:
    Pet();
    Pet(string petName, int yearsOld);
    void Print();

private:
    string name;
    int age;
};

Pet::Pet() {
    name = "Unnamed";
    age = -9999;
}

Pet::Pet(string petName, int yearsOld) {
    name = petName;
    age = yearsOld;
}

void Pet::Print() {
    cout << name << ", " << age << endl;
}

int main() {
    Pet dog;
    Pet cat("Bella", 8);

    cat.Print();
    dog.Print();

    return 0;
}
```

Bella, 8
Unnamed, -9999

1

2

3

Check**Next**
CHALLENGE ACTIVITY

2.11.2: Constructor overloading.

489394.3384924.qx3zqy7

Start

The Member class has a default constructor, a constructor with one parameter, and a constructor with the following objects:

- member1 with no arguments
- member2 with memberName as an argument
- member3 with memberName, memberAge, and memberHeight as arguments

©zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Ex: If the input is Kai 24 4.50, then the output is:

```
Member: Unnamed, 0, 0.00
Member: Kai, 0, 0.00
Member: Kai, 24, 4.50
```

```
1 #include <iostream>
2 #include <string>
3 #include <iomanip>
4 using namespace std;
5
6 class Member {
7     public:
8         Member();
9         Member(string memberName);
10        Member(string memberName, int memberAge, double memberHeight);
11        void Print();
12
13    private:
14        string name;
15        int age;
```

©zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

1

2

Check

Next level

2.12 Constructor initializer lists

A **constructor initializer list** is an alternative approach for initializing data members in a constructor, coming after a colon and consisting of a comma-separated list of variableName(initValue) items.

Figure 2.12.1: Member initialization: (left) Using statements in the constructor, (right) Using a constructor initializer list.

©zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

```
#include <iostream>
using namespace std;

class SampleClass {
public:
    SampleClass();
    void Print() const;

private:
    int field1;
    int field2;
};

SampleClass::SampleClass() {
    field1 = 100;
    field2 = 200;
}

void SampleClass::Print() const
{
    cout << "Field1: " << field1
    << endl;
    cout << "Field2: " << field2
    << endl;
}

int main() {
    SampleClass myClass;
    myClass.Print();
    return 0;
}
```

Field1: 100
Field2: 200

```
#include <iostream>
using namespace std;

class SampleClass {
public:
    SampleClass();
    void Print() const;

private:
    int field1;
    int field2;
};

SampleClass::SampleClass() :
    field1(100), field2(200) {}

void SampleClass::Print() const {
    cout << "Field1: " << field1 <<
    endl;
    cout << "Field2: " << field2 <<
    endl;
}

int main() {
    SampleClass myClass;
    myClass.Print();
    return 0;
}
```

@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

2.12.1: Member initialization.

- Convert this constructor to use a constructor initializer list.

```
MyClass::MyClass() {
    x = -1;
    y = 0;
}
```

```
MyClass::MyClass() 
```

Check

Show answer

@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

The approach is important when a data member is a class type that must be explicitly constructed. Otherwise, that data member is by default constructed. Ex: If you have studied vectors, consider a data member consisting of a vector of size 2.

Figure 2.12.2: Member initialization in a constructor.

```
#include <iostream>
#include <vector>
using namespace std;

class SampleClass {
public:
    SampleClass();
    void Print() const;

private:
    vector<int> itemList;
};

SampleClass::SampleClass() {
    // itemList gets default
constructed, size 0
    itemList.resize(2);
}

void SampleClass::Print() const {
    cout << "Size: " <<
itemList.size() << endl;
    cout << "Item1: " <<
itemList.at(0) << endl;
    cout << "Item2: " <<
itemList.at(1) << endl;
}

int main() {
    SampleClass myClass;
    myClass.Print();
    return 0;
}
```

Size: 2
Item1: 0
Item2: 0

```
#include <iostream>
#include <vector>
using namespace std;

class SampleClass {
public:
    SampleClass();
    void Print() const;

private:
    vector<int> itemList;
};

SampleClass::SampleClass() :
itemList(2) {
    // itemList gets constructed
with size 2
}

void SampleClass::Print() const {
    cout << "Size: " <<
itemList.size() << endl;
    cout << "Item1: " <<
itemList.at(0) << endl;
    cout << "Item2: " <<
itemList.at(1) << endl;
}

int main() {
    SampleClass myClass;
    myClass.Print();
    return 0;
}
```

@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

On the left, the constructor initially creates a vector of size 0, then resizes to size 2, where each element has the value 0. On the right, itemList(2) is provided in the SampleClass constructor initialization list, causing the vector constructor to be called with size 2 and each vector element to be initialized with the value 0. Using the initialization list avoids the inefficiency of constructing and then modifying an item.

Note: Since C++11, the data member could have been initialized in the class definition: `vector<int> itemList(2);`. However, initialization lists are still useful for other cases.

PARTICIPATION ACTIVITY

2.12.2: Constructor initializer list.



Consider the example above.

- 1) On the left, itemList is first constructed with size 0, then resized to size 2.

- True
- False

- 2) On the right, itemList is first constructed with size 0, then resized to size 2.

- True

@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



False**CHALLENGE ACTIVITY**

2.12.1: Enter the output of constructor initializer lists.



489394.3384924.qx3zqy7

Start

Type the program's output

```
#include <iostream>
#include <string>
using namespace std;

class Tutor {
public:
    Tutor();
    void Print() const;

private:
    string name;
    string topic;
};

Tutor::Tutor() : name("NeedsName"), topic("MissingTopic") {}

void Tutor::Print() const {
    cout << topic << ", by " << name << endl;
}

int main() {
    Tutor myTutor;

    myTutor.Print();

    return 0;
}
```

MissingTopic, b

1

2

3

4

Check**Next****CHALLENGE ACTIVITY**

2.12.2: Constructor initializer lists.



489394.3384924.qx3zqy7

Start

Add a constructor initializer list to the default City constructor to initialize name with "Empty", population with 0, and tourism with 'X'.

Ex: If the input is Natrona 2445 Y, then the output is:

```
City: Empty, Population: -999, Tourism: X
City: Natrona, Population: 2445, Tourism: Y
```

©zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

```
1 #include <iostream>
2 using namespace std;
3
4 class City {
5 public:
6     City();
7     void SetDetails(string newName, int newPopulation, char newTourism);
8 }
```

```
8     void Print() const;
9     private:
10    string name;
11    int population;
12    char tourism;
13 };
14
15 City::City() /* Your code goes here */ {
```

1

2

@zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

[Check](#)[Next level](#)

Exploring further:

- [Classes](#) from cplusplus.com, see "Member initialization in constructors" section.
- [Constructors](#) from msdn.microsoft.com, see "Member lists".

2.13 The 'this' implicit parameter

Implicit parameter

An object's member function is called using the syntax `object.Function()`. The object variable before the function name is known as an **implicit parameter** of the member function because the compiler converts the call syntax `object.Function(...)` into a function call with a pointer to the object implicitly passed as a parameter. Ex: `Function(object, ...)`.

Within a member function, the implicitly-passed object pointer is accessible via the name **this**. In particular, a member can be accessed as `this->member`. The `->` is the member access operator for a pointer, similar to the `".` operator for non-pointers.

Using `this->` makes clear that a class member is being accessed and is essential if a data member and parameter have the same identifier. In the example below, `this->` is necessary to differentiate between the data member `sideLength` and the parameter `sideLength`.

Figure 2.13.1: Using 'this' to refer to an object's member.

@zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

```
#include <iostream>
using namespace std;

class ShapeSquare {
public:
    void SetSideLength(double sideLength);
    double GetArea() const;
private:
    double sideLength;
};

void ShapeSquare::SetSideLength(double sideLength) {
    this->sideLength = sideLength;
    // Data member      Parameter
}

double ShapeSquare::GetArea() const{
    return sideLength * sideLength; // Both refer to data
member
}

int main() {
    ShapeSquare square1;

    square1.SetSideLength(1.2);
    cout << "Square's area: " << square1.GetArea() << endl;

    return 0;
}
```

@zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Square's area:
1.44

**PARTICIPATION
ACTIVITY**

2.13.1: The 'this' implicit parameter.



Given a class Spaceship with private data member numYears and public member function:

`void Spaceship::AddNumYears(int numYears)`

1) In AddNumYears(), which line assigns the data member numYears with 0?



- `numYears = 0;`
- `this.numYears = 0;`
- `this->numYears = 0;`

2) In AddNumYears(), which line assigns the data member numYears with the parameter numYears?



- `numYears = this->numYears;`
- `this->numYears = numYears;`

3) In AddNumYears(), which line adds the parameter numYears to the existing value of data member numYears?



- `this->numYears = this->numYears + numYears;`
- `this->numYears = numYears + numYears;`

@zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

numYears = this->numYears + numYears;

- 4) Given variable `Spaceship ss1` is declared in `main()`, which line assigns `ss1`'s `numYears` with 5?

`ss1.numYears = 5;`

`ss1->numYears = 5;`

`this->numYears = 5;`

None of the above.

@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Using 'this' in class member functions and constructors

The animation below illustrates how member functions work. When an object's member function is called, the object's memory address is passed to the function via the implicit "this" parameter. An access in `SetTime()` to `this->hours` first goes to the object's address, then to the hours data member. If `SetTime()` instead had the assignment `hours = timeHr`, the compiler would use `this->hours` for `hours` because no other variable in `SetTime()` is named `hours`.

PARTICIPATION
ACTIVITY

2.13.2: How a member function works.

Animation captions:

1. `travTime` is an object of class type `ElapsedTime`.
2. When `travTime`'s `SetTime()` member function is called, `travTime`'s memory address is passed to the function via the implicit "this" parameter.
3. The implicitly-passed object pointer is accessible within the member function via the name "this". Ex: `this->hours` first goes to `travTime`'s address, then to the `hours` data member.

PARTICIPATION
ACTIVITY

2.13.3: Using the 'this' pointer in member functions and constructors.

- 1) Complete the code to assign the value of `mins` to data member `minutes` using `this->` notation.

```
void ElapsedTime::SetMinutes(int mins) {
     = mins;
}
```

Check

Show answer

@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 2) Complete the code to assign the value of parameter `hours` to data member `hours` using `this->` notation.

```
void ElapsedTime::SetHours(int
hours) {
    ;
```

Check**Show answer**

Exploring further:

- [The 'this' pointer](#) from msdn.microsoft.com.

@zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

CHALLENGE ACTIVITY

2.13.1: Enter the output of the function.



489394.3384924.qx3zqy7

Start

Type the program's output

```
#include <iostream>
using namespace std;

class Airplane {
public:
    Airplane();
    void Print() const;
    void SetSpeed(int speed);
private:
    int speed;
};

Airplane::Airplane() {
    speed = 0;
}

void Airplane::SetSpeed(int speed) {
    this->speed = speed;
}

void Airplane::Print() const {
    cout << speed << " MPH" << endl;
}

int main() {
    Airplane boeing747;

    boeing747.SetSpeed(650);
    boeing747.Print();

    return 0;
}
```



1

2

@zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Check**Next****CHALLENGE ACTIVITY**

2.13.2: The this implicit parameter.



Define the missing member function. Use "this" to distinguish the local member from the parameter name.

[Learn how our autograder works](#)

489394.3384924.qx3zqy7

```

1 #include <iostream>
2 using namespace std;
3
4 class CablePlan{
5     public:
6         void SetNumDays(int numDays);
7         int GetNumDays() const;
8     private:
9         int numDays;
10 };
11
12 // FIXME: Define SetNumDays() member function, using "this" implicit parameter
13 void CablePlan::SetNumDays(int numDays) {
14
15     /* Your solution goes here */
16 }
```

©zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Run

View your last submission ▾

2.14 Operator overloading

Overview

C++ allows a programmer to redefine the functionality of built-in operators like +, -, and *, to operate on programmer-defined objects, a process known as **operator overloading**. Suppose a class TimeHrMn has data members hours and minutes. Overloading + would allow two TimeHrMn objects to be added with the + operator.

PARTICIPATION ACTIVITY

2.14.1: Operator overloading allows use of operators like + on classes.



Animation content:

undefined

Animation captions:

- timeTot is initialized to the sum of time1 and time2 by adding the hours and minutes fields separately.
- Overloading the + operator in the TimeHrMn class allows the time1 and time2 objects to be added directly.
- The same result is achieved with simpler, more readable code.

PARTICIPATION ACTIVITY

2.14.2: Operator overloading.



Refer to the example above.

- 1) The expression `time1 + time2`
results in a compiler error if the +
operator is not overloaded inside the
`TimeHrMn` class.

- True
- False

- 2) The expressions `time1 + time2` and
`time2 + time1` are expected to
produce the same result.

- True
- False

©zyBooks 05/27/23 21:58 1692462

Taylor Larrechea
COLORADOCSPB2270Summer2023

Overloading `TimeHrMn`'s + operator

To overload +, the programmer creates a member function named `operator+`. Although + requires left and right operands as in `time1 + time2`, the member function only requires the right operand (rhs: right-hand-side) as the parameter, because the left operand is the calling object. In other words, `time1 + time2` is equivalent to the function call `time1.operator+(time2)`, which is valid syntax but almost never used.

Figure 2.14.1: `TimeHrMn` class implementation with overloaded + operator.

```
#include <iostream>
using namespace std;

class TimeHrMn {
public:
    TimeHrMn(int timeHours = 0, int timeMinutes = 0);
    void Print() const;
    TimeHrMn operator+(TimeHrMn rhs);
private:
    int hours;
    int minutes;
};

// Overload + operator for TimeHrMn
TimeHrMn TimeHrMn::operator+(TimeHrMn rhs) {
    TimeHrMn timeTotal;

    timeTotal.hours = hours + rhs.hours;
    timeTotal.minutes = minutes + rhs.minutes;

    return timeTotal;
}

TimeHrMn::TimeHrMn(int timeHours, int timeMinutes) {
    hours = timeHours;
    minutes = timeMinutes;
}

void TimeHrMn::Print() const {
    cout << "H:" << hours << ", " << "M:" << minutes << endl;
}
```

©zyBooks 05/27/23 21:58 1692462

Taylor Larrechea
COLORADOCSPB2270Summer2023

**PARTICIPATION
ACTIVITY**

2.14.3: TimeHrMn::operator+ is called when two TimeHrMn objects are added with the + operator.

**Animation content:**

undefined

Animation captions:

@zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

COLORADO CSPB2270Summer2023

1. time1, time2, and sumTime are initialized. sumTime initially has 0 hours and 0 minutes.
2. The expression time1 + time2 results in TimeHrMn's operator+ member function being called.
3. In the expression hours + rhs.hours, hours is time1's hours, and rhs.hours is time2's hours.
4. The minutes are computed similarly. The combined time is returned and displayed.

**PARTICIPATION
ACTIVITY**

2.14.4: Operator overloading basics.



- 1) Given `TimeHrMn time1(10, 0)`
 and `TimeHrMn time2(3, 5)`, and
 the above overloading of +, what is
`sumTime.hours` after `sumTime =`
`time1 + time2?`

Check**Show answer**

- 2) Write the start of a TimeHrMn member
 function definition that overloads the
 subtraction (-) operator, naming the
 parameter rhs.



```
// Overloaded '-' function
definition
[REDACTED]
{
    /* Implementation */
}
```

Check**Show answer**

- 3) Which parameter should be
 removed from this line, that strives
 to overload the * operator? Type the
 parameter name only; don't list the
 type.

@zyBooks 05/27/23 21:58 1692462
 Taylor Larrechea
 COLORADO CSPB2270Summer2023

```
TimeHrMn
TimeHrMn::operator*
(TimeHrMn lhs, TimeHrMn
rhs) {
```

Check**Show answer**

Overloading the + operator multiple times

When an operator like + has been overloaded, the compiler determines which + operation to invoke based on the operand types. In `4 + 9`, the compiler sees two integer operands and thus applies the built-in + operation. In `time1 + time2`, where `time1` and `time2` are `TimeHrMn` objects, the compiler sees two `TimeHrMn` operands and thus invokes the programmer-defined function.

©zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

A programmer can define several functions that overload the same operator, as long as each involves different types so that the compiler can determine which to invoke. The code below overloads the + operator twice in the `TimeHrMn` class.

`main()` uses the + operator in 4 statements. The first + involves two `TimeHrMn` operands, so the compiler invokes the first operator+ function ("A"). The second + involves `TimeHrMn` and `int` operands, so the compiler invokes the second operator+ function ("B"). The third + involves two `int` operands, so the compiler invokes the built-in + operation. The fourth +, commented out, involves an `int` and `TimeHrMn` operands. Because no function has those operands ("B" has `TimeHrMn` and `int`, not `int` and `TimeHrMn`; order matters), that statement would generate a compiler error.

Figure 2.14.2: Overloading the + operator multiple times.

©zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

```
#include <iostream>
using namespace std;

class TimeHrMn {
public:
    TimeHrMn(int timeHours = 0, int timeMinutes = 0);
    void Print() const;
    TimeHrMn operator+(TimeHrMn rhs);
    TimeHrMn operator+(int rhsHours);
private:
    int hours;
    int minutes;
};

// Operands: TimeHrMn, TimeHrMn. Call this "A"
TimeHrMn TimeHrMn::operator+(TimeHrMn rhs) {
    TimeHrMn timeTotal;

    timeTotal.hours = hours + rhs.hours;
    timeTotal.minutes = minutes + rhs.minutes;

    return timeTotal;
}

// Operands: TimeHrMn, int. Call this "B"
TimeHrMn TimeHrMn::operator+(int rhsHours) {
    TimeHrMn timeTotal;

    timeTotal.hours = hours + rhsHours;
    timeTotal.minutes = minutes; // Stays same

    return timeTotal;
}

TimeHrMn::TimeHrMn(int timeHours, int timeMinutes) {
    hours = timeHours;
    minutes = timeMinutes;

    return;
}

void TimeHrMn::Print() const {
    cout << "H:" << hours << ", " << "M:" << minutes <<
endl;
}

int main() {
    TimeHrMn time1(3, 22);
    TimeHrMn time2(2, 50);
    TimeHrMn sumTime;
    int num;

    num = 91;

    sumTime = time1 + time2; // Invokes "A"
    sumTime.Print();

    sumTime = time1 + 10; // Invokes "B"
    sumTime.Print();

    cout << num + 8 << endl; // Invokes built-in add

    // timeTot = 10 + time1; // ERROR: No (int, TimeHrMn)

    return 0;
}
```

H:5, M:72
H:13,
M:22
99

@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

**PARTICIPATION
ACTIVITY**

2.14.5: Determining which function is invoked.

Given:

```
Course course1;
Course course2;
int num1;
int num2;
```

If unable to drag and drop, refresh the page.

num1 + num2;**course1 + course2;****course1 + num1;****num2 + course2;**

©zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Error

Course Course::operator+(int val) {

Course Course::operator+(Course rhs) {

Built-in + operation

Reset**CHALLENGE
ACTIVITY**

2.14.1: Enter the output of operator overloading.

489394.3384924.qx3zqy7

Start

Type the program's output

©zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

```
#include <iostream>
using namespace std;

class InchSize {
public:
    InchSize(int wholeInches = 0, int sixteenths = 0);
    void Print() const;
    InchSize operator+(InchSize rhs);
private:
    int inches;
    int sixteenths;
};

InchSize InchSize::operator+(InchSize rhs) {
    InchSize totalSize;

    totalSize.inches = inches + rhs.inches;
    totalSize.sixteenths = sixteenths + rhs.sixteenths;

    // If sixteenths is greater than an inch, carry 1 to inches.
    if (totalSize.sixteenths >= 16) {
        totalSize.inches += 1;
        totalSize.sixteenths -= 16;
    }

    return totalSize;
}

InchSize::InchSize(int wholeInches, int sixteenthsOfInch) {
    inches = wholeInches;
    sixteenths = sixteenthsOfInch;
}

void InchSize::Print() const {
    cout << inches << " " << sixteenths << "/16 inches" << endl;
}

int main() {
    InchSize size1(5, 8);
    InchSize size2(8, 11);
    InchSize sumSize;

    sumSize = size1 + size2;

    sumSize.Print();

    return 0;
}
```

©zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



1

2

Check**Next**
CHALLENGE ACTIVITY

2.14.2: Operator overloading.



489394.3384924.qx3zqy7

Start

Four doubles are read from input, where the first two doubles are the length and width of rectangle1 and the last two doubles are the length and width of rectangle2. Complete the function to overload the + operator.

Ex: If the input is 14.5 15.5 4.0 7.0, then the output is:

Length: 14.5 units, width: 15.5 units
Length: 4 units, width: 7 units
Sum: Length: 18.5 units, width: 22.5 units

Note: The sum of two rectangles is:

- the sum of the lengths of the rectangles
- the sum of the widths of the rectangles

©zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

```

1 #include <iostream>
2 using namespace std;
3
4 class Rectangle {
5     public:
6         Rectangle(double numLength = 0.0, double numWidth = 0.0);
7         void Print() const;
8         Rectangle operator+(Rectangle rhs);
9     private:
10        double length;
11        double width;
12    };
13
14 Rectangle::Rectangle(double numLength, double numWidth) {
15     length = numLength;
16     width = numWidth;
17 }
```

@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

1

2

Check**Next level**

Exploring further:

- [Overloadable operators](#) from cplusplus.com. Provides a list of operators that can be overloaded, including a description of how to declare overloaded operator functions for operators with different operands like += and ++.

2.15 Overloading comparison operators

Overloading the equality (==) operator

A programmer can overload the equality operator (==) to allow comparing objects of a programmer-defined class for equality. To overload ==, the programmer creates a function named `operator==` that returns `bool` and takes two `const` reference arguments of the class type for the left-hand-side and right-hand-side operands. Ex: To overload the == operator for a `Review` class, the programmer defines a function `bool operator==(const Review& lhs, const Review& rhs)`.

The programmer must also determine when two objects are considered equal. In the `Review` class below, two `Review` objects are equal if the objects have the same rating and comment.

PARTICIPATION ACTIVITY

2.15.1: Overloading the == operator.

@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Animation content:

undefined

Animation captions:

1. `myReview` is the left operand of the == operator and is passed as the first argument.
`bestReview` is the right operand and is passed as the second argument.

2. The operator== function returns true if both operands have the same rating and comment. myReview and bestReview both have a rating of 5 and a comment of "Great", so the operator returns true.

PARTICIPATION ACTIVITY**2.15.2: Overloading == operator for Restaurant class.**

Given a Restaurant class, which of the following are valid function signatures for overloading the equality (==) operator?

©zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



1) `operator==(const Restaurant& lhs, const Restaurant& rhs)`

- Valid
- Invalid



2) `bool operator==(const Restaurant lhs, const Restaurant rhs)`

- Valid
- Invalid



3) `bool operator==(const Restaurant& lhs, const string& rhs)`

- Valid
- Invalid

Overloading the < operator

A programmer can also overload relational operators like the less than operator (<). The < operator should return true if the object on the left side of the < operator is less than the object on the right side of the operator. In the Review class below, the `operator<` function returns true if the left Review operand has a lower rating than the right Review operand.

Figure 2.15.1: Overloading the Reviews class' < operator.

©zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

class Review {
public:
    void SetRatingAndComment(int revRating, string revComment) {
        rating = revRating;
        comment = revComment;
    }
    int GetRating() const { return rating; }
    string GetComment() const { return comment; }

private:
    int rating = -1;
    string comment = "NoComment";
};

// Equality (==) operator for two Review objects
bool operator==(const Review& lhs, const Review& rhs) {
    return (lhs.GetRating() == rhs.GetRating()) &&
           (lhs.GetComment() == rhs.GetComment());
}

// Less-than (<) operator for two Review objects
bool operator<(const Review& lhs, const Review& rhs) {
    return lhs.GetRating() < rhs.GetRating();
}

int main() {
    vector<Review> reviewList;
    Review currentReview;
    Review lowestReview;
    int currentRating;
    string currentComment;
    int i;

    cout << "Type rating + comments. To end: -1" << endl;
    cin >> currentRating;
    while (currentRating >= 0) {
        getline(cin, currentComment); // Gets rest of line
        currentReview.SetRatingAndComment(currentRating,
        currentComment);
        reviewList.push_back(currentReview);
        cin >> currentRating;
    }

    // Find and output lowest review
    lowestReview = reviewList.at(0);
    for (i = 1; i < reviewList.size(); ++i) {
        if (reviewList.at(i) < lowestReview ) {
            lowestReview = reviewList.at(i);
        }
    }

    cout << endl;
    cout << lowestReview.GetRating() << " "
        << lowestReview.GetComment() << endl;

    return 0;
}
```

@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

```
Type rating + comments. To end: -1
5 Great place!
5 Loved the food.
2 Pretty bad service.
4 New owners are nice.
2 Yuk!!!
4 What a gem.
-1

2 Pretty bad service.
```

©zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

2.15.3: Overloading the < operator.

Given the Review class above, complete the definition for the overloaded < operator for the given comparison types. Use const reference parameters, and name the operands lhs and rhs.

1) Review < int

```
bool operator<(const Review&
lhs, [REDACTED]) {
    return lhs.GetRating() <
rhs;
}
```

Check**Show answer**

2) int < Review

```
bool operator<(const int& lhs,
const Review& rhs) {
    return lhs <
[REDACTED];
}
```

Check**Show answer**

3) Review < double

```
[REDACTED]
{
    return lhs.GetRating() < rhs;
}
```

Check**Show answer**

Overloading all equality and relational operators

A common approach is to first overload the == and < operators and then overload other comparison operators using == and <.

©zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

- Overloading != using ==:

```
bool operator!=(const Review& lhs, const Review& rhs) { return !(lhs == rhs); }
```

- Overloading >, <=, and >= using <:

```
bool operator>(const Review& lhs, const Review& rhs) { return rhs < lhs; }
bool operator<=(const Review& lhs, const Review& rhs) { return !(lhs > rhs); }
bool operator>=(const Review& lhs, const Review& rhs) { return !(lhs < rhs); }
```

**PARTICIPATION
ACTIVITY****2.15.4: Overloading comparison operators.**

Given two Review objects named userReview and bestReview, which overloaded operators are called for the following comparisons?

1) `userReview != bestReview`



- operator==
- operator!=
- operator!= and operator==

©zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

2) `userReview > bestReview`



- operator<
- operator>
- operator> and operator<

3) `userReview <= bestReview`



- operator<= and operator==
- operator<=, operator>, and operator==;
- operator<=, operator>, and operator<

Sorting a vector

The **sort()** function, defined in the C++ Standard Template Library's (STL) algorithms library, can sort vectors containing objects of programmer-defined classes. To use `sort()`, a programmer must:

1. Add `#include <algorithm>` to enable the use of `sort()`.
2. Overload the `<` operator for the programmer-defined class.
3. Call the `sort()` function as `sort(myVector.begin(), myVector.end())`

Figure 2.15.2: Sorting a vector of Review objects.

©zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
using namespace std;

class Review {
public:
    void SetRatingAndComment(int revRating, string revComment) {
        rating = revRating;
        comment = revComment;
    }
    int GetRating() const { return rating; }
    string GetComment() const { return comment; }

private:
    int rating = -1;
    string comment = "NoComment";
};

// Less-than (<) operator for two Review objects
bool operator<(const Review& lhs, const Review& rhs) {
    return lhs.GetRating() < rhs.GetRating();
}

int main() {
    vector<Review> reviewList;
    Review currentReview;
    int currentRating;
    string currentComment;
    int i;

    cout << "Type rating + comments. To end: -1" << endl;
    cin >> currentRating;
    while (currentRating >= 0) {
        getline(cin, currentComment); // Gets rest of line
        currentReview.SetRatingAndComment(currentRating,
        currentComment);
        reviewList.push_back(currentReview);
        cin >> currentRating;
    }

    // Sort reviews from lowest to highest
    sort(reviewList.begin(), reviewList.end());

    cout << endl;
    for (i = 0; i < reviewList.size(); ++i) {
        cout << reviewList.at(i).GetRating() << ":" <<
        reviewList.at(i).GetComment() << endl;
    }

    return 0;
}
```

Type rating + comments. To end: -1
5 Great place!
5 Loved the food.
2 Pretty bad service.
4 New owners are nice.
2 Yuk!!!
4 What a gem.
-1
2: Pretty bad service.
2: Yuk!!!
4: New owners are nice.
4: What a gem.
5: Great place!
5: Loved the food.

@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

**PARTICIPATION
ACTIVITY**

2.15.5: Sorting vectors.



1) Sorting a vector of integers requires overloading the less than operator for two int operands.

 True False

2) If a vector contains duplicate elements, a programmer must overload both the less than operator and the equality operator.

 True False

3) The sort() function can be used to sort a range of elements within a vector instead of the entire vector.

 True False

@zyBooks 05/27/23 21:58 1692462



Taylor Larrechea

COLORADOCSPB2270Summer2023

**CHALLENGE
ACTIVITY**

2.15.1: Enter the output of the program using overloading operators.



489394.3384924.qx3zqy7

Start

Type the program's output

@zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

```
#include <iostream>
#include <string>
using namespace std;

class Movie {
public:
    Movie(string movieTitle);
    void SetUpVotesAndDownVotes(int numUpVotes, int numDownVotes) {
        upVotes = numUpVotes;
        downVotes = numDownVotes;
    }
    string GetTitle() const { return title; }
    int GetUpVotes() const { return upVotes; }
    int GetDownVotes() const { return downVotes; }

private:
    string title;
    int upVotes;
    int downVotes;
};

Movie::Movie(string movieTitle) {
    title = movieTitle;
    upVotes = 0;
    downVotes = 0;
}

bool operator==(const Movie& movie1, const Movie& movie2) {
    return movie1.GetDownVotes() == movie2.GetDownVotes();
}

int main() {
    Movie movie1("Up");
    Movie movie2("Taken");

    movie1.SetUpVotesAndDownVotes(9, 2);
    movie2.SetUpVotesAndDownVotes(9, 3);

    if (movie1 == movie2) {
        cout << "Equal" << endl;
    }
    else {
        cout << "Not equal" << endl;
    }

    return 0;
}
```

©zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



1

2

3

[Check](#)[Next](#)

2.16 Vector ADT

vector ADT

The **standard template library (STL)** defines classes for common Abstract Data Types (ADTs). A **vector** is an ADT of an ordered, indexable list of items. The vector ADT is implemented as a class (actually a class template that supports different types such as `vector<int>` or `vector<string>`, although templates are discussed elsewhere).

For the commonly-used vector member functions below, assume a vector is declared as:

```
vector<T> vectorName();
```

where T represents the vector's element type, such as:

```
vector<int> teamNums(5);
```

Table 2.16.1: Vector ADT functions.

Notes: size_type is an unsigned integer type. T represents the vector's element type.

at()	<code>at(size_type n)</code> Accesses element n.	<code>teamNums.at(3) = Assigns 99 to e. x = teamNums.at(3) = Assigns element x @zyBooks 05/27/23 21:58 1692462</code> Taylor Larrechea
size()	<code>size_type size() const;</code> Returns vector's size.	<code>COLORADOCSPB2270Summer2023 if (teamNums.size() == 5) { Size is 5 so condition ... }</code>
empty()	<code>bool empty() const;</code> Returns true if size is 0.	<code>if (teamNums.empty()) { Is 5 so condition ... }</code>
clear()	Removes all elements. Vector size becomes 0.	<code>teamNums.clear(); Vector now has 0 elements. cout << teamNums[0]; Prints 0 teamNums.at(3) = 99; // Error; element at index 3 does not exist</code>
push_back()	<code>void push_back(const T& x);</code> Copies x to new element at vector's end, increasing size by 1. Parameter is pass by reference to avoid making local copy, but const to make clear not changed.	<code>// Assume vector is empty. teamNums.push_back(77); Vector is: 77 teamNums.push_back(2); Vector is: 77, 2 cout << teamNums[1]; Prints 2</code>
erase()	<code>iterator erase (iteratorPosition);</code> Removes element from position. Elements from higher positions are shifted back to fill gap. Vector size decrements.	<code>// Assume vector is [77, 2] teamNums.erase(teamNums.begin() + 1); // Now 77, empty // (Strange position explained below)</code>
insert()	<code>iterator insert(iteratorPosition, const T& x);</code> Copies x to element at position. Items at that position and higher are shifted over to make room. Vector size increments.	<code>// Assume vector is empty. teamNums.insert(teamNums.begin() + 1, 33); // Now [33]</code>

Basic vector functions

@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Use of at(), size(), empty(), and clear() should be straightforward.

PARTICIPATION ACTIVITY

2.16.1: Vector functions at(), size(), empty(), and clear().



Given `vector<int> itemList(10);` Assume all elements have been assigned 0.

- 1) itemList().size returns 10.



- True
- False

2) itemList.size(10) returns 10.



- True
- False

3) itemList.size() returns 10.



@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- True
- False

4) itemList.at(10) returns 0.



- True
- False

5) itemList.empty() removes all elements.



- True
- False

6) After itemList.clear(), itemList.at(0) is an invalid access.



- True
- False

vector's push_back() function

push_back() appends an item to the vector's end, automatically resizing the vector.

PARTICIPATION ACTIVITY

2.16.2: Vector push_back() member function.



Animation captions:

1. When initially declared, the vector vctr has a size of 0.
2. The push_back() function appends a new element to the vector's end and automatically resizes the vector.

One can deduce that the vector class has a private data member that stores the current size. In fact, the vector class has several private data members. However, to use a vector, a programmer only needs to know the public abstraction of the vector ADT.

@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Example: List of players' jersey numbers

The program below assists a soccer coach scouting players, allowing the coach to enter the jersey number of players, and printing a list of those numbers when requested.

The line highlighted in the PlayersAdd() function illustrates use of the push_back() member method. Note from the sample input/output that the items are stored in the vector in the order the items were added. Note that the programmer did not specify an initial vector size in main(), meaning the initial size is 0.

Figure 2.16.1: Using vector member functions: A player jersey numbers program.

```
#include <iostream>
#include <vector>
using namespace std;

// Adds playerNum to end of vector
void PlayersAdd(vector<int>& players, int
playerNum) {
    players.push_back(playerNum);
}

void PlayersPrint(const vector<int>& players) {
    unsigned int i;

    for (i = 0; i < players.size(); ++i) {
        cout << " " << players.at(i) << endl;
    }
}

// Maintains vector of player numbers
int main() {
    vector<int> players;
    int playerNum;
    char userKey;

    userKey = '?';

    cout << "Commands: 'a' add, 'p' print" << endl;
    cout << " 'q' quits" << endl;
    while (userKey != 'q') {
        cout << "Command: ";
        cin >> userKey;
        if (userKey == 'a') {
            cout << " Player number: ";
            cin >> playerNum;
            PlayersAdd(players, playerNum);
        }
        else if (userKey == 'p') {
            PlayersPrint(players);
        }
    }

    return 0;
}
```

©zyBooks 05/27/23 21:58 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

```
Commands: 'a' add, 'p'
print
'q' quits
Command: p
Command: a
Player number: 23
Command: a
Player number: 47
Command: p
23
47
Command: a
Player number: 19
Command: p
23
47
19
Command: q
```

PARTICIPATION ACTIVITY

2.16.3: push_back() function.



Given: `vector<int> itemList;`

If appropriate, type: Error

Answer the questions in order; each may modify the vector.

- What is the initial vector's size?

Check

[Show answer](#)

©zyBooks 05/27/23 21:58 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

- After `itemList.at(0) = 99`, what is the vector's size?



- 3) After itemList.push_back(99), what is the vector's size?



@zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

- 4) After itemList.push_back(77), what are the vector's contents? Type element values in order separated by one space as in: 44 66



- 5) After itemList.push_back(44), what is the vector's size?



- 6) What does itemList.at(itemList.size()) return?



vector's insert() and erase() member functions

The insert() function takes a position argument indicating where the new element should be inserted. However, position is not just a number like 1, but is rather: myVector.begin() + 1. The reason is beyond our scope here, but has to do with *iterators* that can be useful when iterating through a vector in a loop. The erase() function is similar.

PARTICIPATION
ACTIVITY

2.16.4: insert() and erase() vector member functions.



Animation captions:

@zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

1. The erase() function takes a position argument indicating where the element should be removed. Elements from higher positions are shifted back. The vector size is automatically decremented.
2. The insert() function takes a position argument indicating where the element should be added and the element's value. Elements in that position and higher positions are shifted forward. The vector size is automatically incremented.
3. Note that the position argument is not an integer, but an iterator. The begin() function returns an iterator pointing to the first element of the vector. Then, an integer value is added to indicate the

desired element's position.

Example: Players' jersey numbers program with delete option

The `erase()` function can be used to extend the player jersey numbers program with a player delete option, as shown below.

The program's `PlayersDelete()` function uses a common while loop form for finding an item in a vector. The loop body checks if the current item is a match. If so, the item is deleted using the `erase()` function, and the variable `found` is set to true. The loop expression exits the loop if `found` is true, since no further search is necessary. A while loop is used rather than a for loop because the number of iterations is not known beforehand.

Taylor Larrechea
COLORADOCSPB2270Summer2023

Figure 2.16.2: Using the vector `erase()` function.

```
#include <iostream>
#include <vector>
using namespace std;

// Adds playerNum to end of vector
void PlayersAdd(vector<int>& players, int playerNum) {
    players.push_back(playerNum);
}

void PlayersPrint(const vector<int>& players) {
    unsigned int i;

    for (i = 0; i < players.size(); ++i) {
        cout << " " << players.at(i) << endl;
    }
}

// Deletes playerNum from vector
void PlayersDelete(vector<int>& players, int playerNum) {
    unsigned int i = 0;
    bool found = false;

    // Search for playerNum in vector
    while (!found && (i < players.size())) {
        if (players.at(i) == playerNum) {
            players.erase(players.begin() + i); // Delete
            found = true;
        }
        ++i;
    }
}

// Maintains vector of player numbers
int main() {
    vector<int> players;
    int playerNum;
    char userKey;

    userKey = '?';

    cout << "Commands: 'a' add, 'p' print, 'd' del"
<< endl;
    cout << "    'q' quits" << endl;
    while (userKey != 'q') {
        cout << "Command: ";
        cin >> userKey;
        if (userKey == 'a') {
            cout << " Player number: ";
            cin >> playerNum;
            PlayersAdd(players, playerNum);
        }
        else if (userKey == 'p') {
            PlayersPrint(players);
        }
        else if (userKey == 'd') {
            cout << " Player number: ";
            cin >> playerNum;
            PlayersDelete(players, playerNum);
        }
    }

    return 0;
}
```

@zyBooks 05/27/23 21:58 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

```
Commands: 'a' add, 'p'
print, 'd' del
'q' quits
Command: a
Player number: 23
Command: a
Player number: 47
Command: a
Player number: 19
Command: p
23
47
19
Command: d
Player number: 23
Command: p
47
19
Command: d
Player number: 19
Command: p
47
Command: q
```

@zyBooks 05/27/23 21:58 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

Inserting elements in sorted order

A common use of insert() is to insert a new item in sorted order.

PARTICIPATION ACTIVITY

2.16.5: Intuitive depiction of how to add items to a vector while maintaining items in ascending sorted order.


Animation captions:

1. The first number is added to the vector.
2. 44 is greater than 27 so it is added to the end of the vector.
3. 9 is less than 27. 44 and 27 are moved down so 9 can be added to front of the vector.
4. The rest of the numbers are added in the appropriate spots.

©zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

COLORADO CSPB2270Summer2023

zyDE 2.16.1: Insert in sorted order.

Run the program and observe the output to be: 55 4 250 19. Modify the numsInsert function to insert each item in sorted order. The new program should output: 4 19 55 250

[Load default template](#)

```

1
2 #include <iostream>
3 #include <vector>
4 using namespace std;
5
6 void numsInsert(vector<int>& numsList, int newNum) {
7     unsigned int i;
8
9     for (i = 0; i < numsList.size(); ++i) {
10        if (newNum < numsList.at(i)) {
11            // FIXME: insert newNum at element i
12            break; // Exits the for loop
13        }
14    }
15 }
```

[Run](#)


PARTICIPATION ACTIVITY

2.16.6: The insert() and erase() functions.

Given: `vector<int> itemList;`

Assume itemList currently contains: 33 77 44.

Answer questions in order, as each may modify the vector.

©zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

COLORADO CSPB2270Summer2023

- 1) itemList.at(1) returns 77.

- True
- False



- 2) itemList.insert(itemList.begin() + 1, 55)
changes itemList to:



33 55 77 44.

- True
- False

3) `itemList.insert(itemList.begin() + 0, 99)`
inserts 99 at the front of the list.



- True
- False

4) Assuming `itemList` is 99 33 55 77 44,
then `itemList.erase(itemList.begin() + 55)` results in:
99 33 77 44

©zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- True
- False

5) To maintain a list in ascending sorted order, a given new item should be inserted at the position of the first element that is greater than the item.



- True
- False

6) To maintain a list in descending sorted order, a given new item should be inserted at the position of the first element that is equal to the item.



- True
- False

Exploring further:

- [Vectors](#) at cplusplus.com
- [Vectors](#) at msdn.microsoft.com

**CHALLENGE
ACTIVITY**

2.16.1: Enter the output of the vector ADT functions.



489394.3384924.qx3zqy7

Start

©zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Type the program's output

```
#include <iostream>
#include <vector>
using namespace std;

void PrintSize(vector<int> numsList) {
    cout << numsList.size() << " items" << endl;
}

int main() {
    int currVal;
    vector<int> intList(2);

    PrintSize(intList);

    cin >> currVal;
    while (currVal >= 0) {
        intList.push_back(currVal);
        cin >> currVal;
    }

    PrintSize(intList);

    intList.clear();

    PrintSize(intList);

    return 0;
}
```

Input

1 2 3 -1

@zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Output



1

2

[Check](#)[Next](#)
**CHALLENGE
ACTIVITY**

2.16.2: Modifying vectors.



Modify the existing vector's contents, by erasing the element at index 1 (initially 200), then inserting 100 and 102 in the shown locations. Use Vector ADT's `erase()` and `insert()` only, and remember that the first argument of those functions is special, involving an iterator and not just an integer. Sample output of below program with input 33 200 10:

100 33 102 10

[Learn how our autograder works](#)

489394.3384924.qx3zqy7

```
1 #include <iostream>
2 #include <vector>
3 using namespace std;
4
5 void PrintVectors(vector<int> numsList) {
6     unsigned int i;
7
8     for (i = 0; i < numsList.size(); ++i) {
9         cout << numsList.at(i) << " ";
10    }
11    cout << endl;
12 }
13
14 int main() {
15     vector<int> numsList;
```

@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

[Run](#)

[View your last submission ▾](#)

2.17 Namespaces

Defining a namespace

©zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

A **name conflict** occurs when two or more items like variables, classes, or functions, have the same name. Ex: One programmer creates a Seat class for auditoriums, and a second programmer creates a Seat class for airplanes. A third programmer creating a reservation system for airline and concert tickets wants to use both Seat classes, but a compiler error occurs due to the name conflict.

A **namespace** defines a region (or scope) used to prevent name conflicts. Above, the auditorium seat class code can be put in an **auditorium** namespace, and airplane seat class code in an **airplane** namespace. The **scope resolution operator ::** allows specifying in which namespace to find a name, as in: **auditorium::Seat concertSeat;** and **airplane::Seat flightSeat;**

PARTICIPATION ACTIVITY

2.17.1: Namespaces can resolve name conflicts.

**Animation content:**

undefined

Animation captions:

1. A Seat class is declared in auditorium.h, and another Seat class in airplane.h. The compiler generates an error due to a name conflict.
2. The auditorium Seat class may be put in namespace "auditorium", and airplane seat code in a namespace "airplane".
3. The two kinds of seats can then be declared as auditorium::Seat concertSeat and airplane::Seat flightSeat. The compiler now knows which Seat is which.

PARTICIPATION ACTIVITY

2.17.2: Namespaces.



- 1) Two same-named classes can cause a name conflict, but two same-named functions cannot.

- True
- False

- 2) A namespace helps avoid name conflicts among classes, functions, and other items in a program.

- True
- False

- 3) With namespaces, name conflicts cannot occur.

©zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



- True
- False

std namespace

All items in the C++ standard library are part of the **std** namespace (short for standard). To use classes like string or predefined objects like cout, a programmer can use one of two approaches:

1. **Scope resolution operator (::)**: A programmer can use the scope resolution operator to specify the std namespace before C++ standard library items. Ex: `std::cout << "Hello";` or `std::string userName;`
2. **Namespace directive**: A programmer can add the statement `using namespace std;` to direct the compiler to check the std namespace for any names later in the file that aren't otherwise declared. Ex: For `string userName;`, the compiler will check namespace std for string.

For code clarity, most programming guidelines discourage `using namespace` directives except perhaps for std.

PARTICIPATION ACTIVITY

2.17.3: std namespace.



- 1) Standard library items like classes and functions are part of a namespace named std.

- True
- False

- 2) The namespace directive `using namespace std;` is required in any program.

- True
- False

- 3) Without `using namespace std;`, a programmer can access cout using `std::cout`.

- True
- False

- 4) Without any namespace directive, `cout << num1` causes the compiler to check the std namespace for cout.

- True
- False



CHALLENGE ACTIVITY

2.17.1: Enter the output from the proper namespace.

©zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



489394.3384924.qx3zqy7

Start

Type the program's output

[main.cpp](#) [imperial.h](#) [metric.h](#)

```
#include "imperial.h"
#include "metric.h"
#include <iostream>

int main() {
    metric::BiggerUnit();

    imperial::BiggerUnit();

    return 0;
}
```

@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

1

2

[Check](#)[Next](#)

2.18 Static data members and functions

Static data members

The keyword **static** indicates a variable is allocated in memory only once during a program's execution. Static variables are allocated memory once and reside in the program's static memory region for the entire program. Thus, a static variable retains a value throughout the program.

In a class, a **static data member** is a data member of the class instead of a data member of each class object. Thus, static data members are independent of any class object, and can be accessed without creating a class object.

A static data member is declared inside the class definition, but must also be defined outside the class declaration. Within a class function, a static data member can be accessed just by variable name. A public static data member can be accessed outside the class using the scope resolution operator: `ClassName::variableName`.

PARTICIPATION ACTIVITY

2.18.1: Static data member used to create object ID numbers.



Animation content:

undefined

Animation captions:

1. The `Store` class' static data member `nextId` is declared in the `Store` class declaration.
2. `Store::nextId` must be defined and initialized outside the class declaration. Only one instance of that variable will exist in memory.
3. When a `Store` object is created, memory is allocated for the object's name, type, and id data members, but not the static member `nextId`.
4. The constructor assigns an object's id with `nextId`, and then increments `nextId`. Each time an object is created, `nextId` is incremented, and each object has a unique id.
5. Any class member function can access or mutate a static data member. `nextId` can also be accessed outside the class using the scope resolution operator (`::`).

@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

2.18.2: Static data members.





1) Each constructed class object creates a new instance of a static data member.

- True
- False

2) All static data members can be accessed anywhere in a program.



- True
- False

3) Outside of the class where declared, private static data members can be accessed using dot notation.



- True
- False

©zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Static member functions

A **static member function** is a class function that is independent of class objects. Static member functions are typically used to access and mutate private static data members from outside the class. Since static methods are independent of class objects, the `this` parameter is not passed to a static member function. So, a static member function can only access a class' static data members.

Figure 2.18.1: Static member function used to access a private static data member.

©zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

```
#include <iostream>
#include <string>
using namespace std;

class Store {
public:
    Store(string storeName, string storeType);
    int getId();
    static int getNextId();

private:
    string name = "None";
    string type = "None";
    int id = 0;
    static int nextId; // Declare static member variable
};

Store::Store(string storeName, string storeType) {
    name = storeName;
    type = storeType;
    id = nextId; // Assign object id with nextId

    ++nextId; // Increment nextId for next object to be created
}

int Store::getId() {
    return id;
}

int Store::getNextId() {
    return nextId;
}

int Store::nextId = 101; // Define and initialize static data member

int main() {
    Store store1("Macy's", "Department");
    Store store2("Albertsons", "Grocery");
    Store store3("Ace", "Hardware");

    cout << "Store 1's ID: " << store1.getId() << endl;
    cout << "Store 2's ID: " << store2.getId() << endl;
    cout << "Store 3's ID: " << store3.getId() << endl;
    cout << "Next ID: " << Store::getNextId() << endl;

    return 0;
}
```

```
Store 1's ID: 101
Store 2's ID: 102
Store 3's ID: 103
Next ID: 104
```

©zyBooks 05/27/23 21:58 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY
2.18.3: Static member functions.


©zyBooks 05/27/23 21:58 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

- 1) A static member function is needed to access or mutate a _____ static data member from outside of the class.
 - public
 - private
- 2) The **this** parameter can be used in a static member function to access an object's non-static data members.



- True
- False

**CHALLENGE
ACTIVITY**

2.18.1: Enter the output with static members.



489394.3384924.qx3zqy7

Start

@zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Type the program's output

```
#include <iostream>
#include <string>
using namespace std;

class FoodType {
public:
    FoodType(string foodType);
    static int nextId;
    void Print();
private:
    string type = "None";
    int id = 0;
};

FoodType::FoodType(string foodType) {
    type = foodType;
    id = nextId;

    nextId += 2;
}

void FoodType::Print() {
    cout << type << ":" << id << endl;
}

int FoodType::nextId = 40;

int main() {
    FoodType order1("Sushi");
    FoodType order2("Noodles");
    FoodType order3("Crab");

    order3.Print();

    return 0;
}
```



1

2

Check**Next****2.19 C++ example: Salary calculation with classes**

@zyBooks 05/27/23 21:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

zyDE 2.19.1: Calculate salary: Using classes.

The program below uses a class, TaxTableTools, which has a tax table built in. The main function prompts for a salary, then uses a TaxTableTools function to get the tax rate. The program then calculates the tax to pay and displays the results to the user. Run the program.

with annual salaries of 10000, 50000, 50001, 100001 and -1 (to end the program) and no output tax rate and tax to pay.

1. Modify the TaxTableTools class to use a setter function that accepts a new salary and tax rate table.
2. Modify the program to call the new function, and run the program again, noting the output.

The program's two classes are in separate tabs at the top.

©zyBooks 05/27/23 21:58 1692462

Note: The calculation is inaccurate to how taxes are formally assessed and is a simplification for educational purposes only.

Taylor Larrechea
COLORADOCSPB2270Summer2023

Current file: **IncomeTaxMain.cpp** ▾ Load default template

```

1 #include <iostream>
2 #include <limits>
3 #include <vector>
4 #include <string>
5 #include "TaxTableTools.h"
6
7 int GetInteger(const string userPrompt) {
8     int inputValue;
9
10    cout << userPrompt << ":" << endl;
11    cin >> inputValue;
12
13    return inputValue;
14 }
15

```

10000 50000 50001 100001 -1

Run

zyDE 2.19.2: Calculate salary: Using classes (solution).

A solution to the above problem follows.

Note that the program's two classes are in separate tabs at the top.

Current file: **IncomeTaxMain.cpp** ▾ Load default template

©zyBooks 05/27/23 21:58 1692462

Taylor Larrechea
COLORADOCSPB2270Summer2023

```

1 #include <iostream>
2 #include <limits>
3 #include <vector>
4 #include <string>
5 #include "TaxTableTools.h"
6
7 int GetInteger(const string userPrompt) {
8     int inputValue;
9
10    cout << userPrompt << ":" << endl;
11    cin >> inputValue;
12
13    return inputValue;
14 }
15

```

```

11     cin >> inputValue;
12
13     return inputValue;
14 }
15

```

```
10000 50000 50001 100001 -1
```

Run

©zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

zyDE 2.19.3: Salary calculation: Overloading a constructor.

The program below calculates a tax rate and tax to pay given an annual salary. The program uses a class, TaxTableTools, which has the tax table built in. Run the program with annual salaries of 10000, 50000, 50001, 100001 and -1 (to end the program) and note the output tax rate and tax to pay.

1. Overload the constructor.
 - a. Add to the TaxTableTools class an overloaded constructor that accepts the base salary table and corresponding tax rate table as parameters.
 - b. Modify the main function to call the overloaded constructor with the two tables (vectors) provided in the main function. Be sure to set the nEntries value, too.
 - c. Note that the tables in the main function are the same as the tables in the TaxTableTools class. This sameness facilitates testing the program with the same annual salary values listed above.
 - d. Test the program with the annual salary values listed above.
2. Modify the salary and tax tables
 - a. Modify the salary and tax tables in the main function to use different salary rates and tax rates.
 - b. Use the just-created overloaded constructor to initialize the salary and tax tables.
 - c. Test the program with the annual salary values listed above.

Note: The calculation is inaccurate to how taxes are formally assessed and is a simplification for educational purposes only.

Current
file:

IncomeTaxMain.cpp ▾

Load default template

```

1 #include <iostream>
2 #include <limits>
3 #include <vector>
4 #include <string>
5 #include "TaxTableTools.h"
6 using namespace std;
7
8 int main() {
9     const string PROMPT_SALARY = "\nEnter annual salary (-1 to exit)"
10    int annualSalary;
11    double taxRate;
12    int taxToPay;
13    vector<int> salaryBase(5);
14    vector<double> taxBase(5);
15

```

©zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

10000
50000
50001

Run

©zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

zyDE 2.19.4: Salary calculation: Overloading a constructor (solution).

A solution to the above problem follows.

Note that the program's two classes are in separate tabs at the top.

Current file: **IncomeTaxMain.cpp** ▾ [Load default template](#)

```
1 #include <iostream>
2 #include <limits>
3 #include <vector>
4 #include <string>
5 #include "TaxTableTools.h"
6 using namespace std;
7
8 int main() {
9     const string PROMPT_SALARY = "\nEnter annual salary (-1 to exit)"
10    int annualSalary;
11    double taxRate;
12    int taxToPay;
13    vector<int> salaryBase(5);
14    vector<double> taxBase(5);
15 }
```

10000
50000
50001
100000

Run

[View all posts by admin](#) | [View all posts in category](#)

2.20 C++ example: Domain name availability with classes

zyDE 2.20.1: Domain name availability: Using classes.

The program below uses a class, DomainAvailabilityTools, which includes a table of registered domain names. The main function prompts for domain names until the user presses Enter at the prompt. The domain name is checked against a list of the registered domains in the DomainAvailabilityTools class. If the domain name is not available, the program displays similar domain names.

1. Run the program and observe the output for the given input.
2. Modify the DomainAvailabilityClass's function named GetSimilarDomainNames so some unavailable domain names do not get a list of similar domain names. Run the program again and observe that unavailable domain names with TLDs of .org or .bi not have similar names.

Current file: **DomainAvailabilityMain.cpp** ▾ [Load default template](#)

```

1 #include <iostream>
2 #include <string>
3 #include <cctype>
4 #include "DomainAvailabilityTools.h"
5 using namespace std;
6
7 // ****
8
9 // prompts user string. Returns string.
10 string GetString(string prompt) {
11     string userInput;
12     ...
13     cout << prompt << endl;
14     cin >> userInput;
15 }
```

programming.com
apple.com
oracle.com

Run

zyDE 2.20.2: Domain validation: Using classes (solution).

A solution to the above problem follows.

Current file: **DomainAvailabilityMain.cpp** ▾ [Load default template](#)

```

1 #include <iostream>
2 #include <string>
3 #include <cctype>
4 #include "DomainAvailabilityTools.h"
5 using namespace std;
6
7 // ****
8
9 // Prompts user for input string and returns the string
10 string GetString(string prompt) {
11     string userInput;
12     ...
13     cout << prompt << endl;
14     cin >> userInput;
15 }
```

```
programming.com  
apple.com  
oracle.com
```

Run

©zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

©zyBooks 05/27/23 21:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

3.1 Introduction to algorithms

Algorithms

An **algorithm** describes a sequence of steps to solve a computational problem or perform a calculation. An algorithm can be described in English, pseudocode, a programming language, hardware, etc. A **computational problem** specifies an input, a question about the input that can be answered using a computer, and the desired output.

PARTICIPATION ACTIVITY

3.1.1: Computational problems and algorithms.



Animation captions:

1. A computational problem is a problem that can be solved using a computer. A computational problem specifies the problem input, a question to be answered, and the desired output.
2. For the problem of finding the maximum value in an array, the input is an array of numbers.
3. The problem's question is: What is the maximum value in the input array? The problem's output is a single value that is the maximum value in the array.
4. The FindMax algorithm defines a sequence of steps that determines the maximum value in the array.

PARTICIPATION ACTIVITY

3.1.2: Algorithms and computational problems.



Consider the problem of determining the number of times (or frequency) a specific word appears in a list of words.

- 1) Which can be used as the problem input?

- String for user-specified word
- Array of unique words and string for user-specified word
- Array of all words in the list and string for user-specified word

©zyBooks 05/27/23 22:12 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 2) What is the problem output?

- Integer value for the frequency of most frequent word

- String value for the most frequent word in input array
 - Integer value for the frequency of specified word
- 3) An algorithm to solve this computation problem must be written using a programming language.
- True
 - False

©zyBooks 05/27/23 22:12 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Practical applications of algorithms

Computational problems can be found in numerous domains, including e-commerce, internet technologies, biology, manufacturing, transportation, etc. Algorithms have been developed for numerous computational problems within these domains.

A computational problem can be solved in many ways, but finding the best algorithm to solve a problem can be challenging. However, many computational problems have common subproblems, for which efficient algorithms have been developed. The examples below describe a computational problem within a specific domain and list a common algorithm (each discussed elsewhere) that can be used to solve the problem.

Table 3.1.1: Example computational problems and common algorithms.

Application domain	Computational problem	Common algorithm
DNA analysis	Given two DNA sequences from different individuals, what is the longest shared sequence of nucleotides?	<p><i>Longest common substring problem:</i> A longest common substring algorithm determines the longest common substring that exists in two input strings.</p> <p>DNA sequences can be represented using strings consisting of the letters A, C, G, and T to represent the four different nucleotides.</p>
Search engines	Given a product ID and a sorted array of all in-stock products, is the product in	<i>Binary search:</i> The binary search algorithm is an efficient algorithm for searching a list. The list's elements must be sorted and directly accessible (such as an array).

	stock and what is the product's price?	
Navigation	Given a user's current location and desired location, what is the fastest route to walk to the destination?	<p><i>Dijkstra's shortest path:</i> Dijkstra's shortest path algorithm determines the shortest path from a start vertex to each vertex in a graph.</p> <p>©zyBooks 05/27/23 22:12 1692462 Taylor Larrechea</p> <p>The possible routes between two locations can be represented using a graph, where vertices represent specific locations and connecting edges specify the time required to walk between those two locations.</p>

PARTICIPATION ACTIVITY

3.1.3: Computational problems and common algorithms.



Match the common algorithm to another computational problem that can be solved using that algorithm.

If unable to drag and drop, refresh the page.

Binary search**Shortest path algorithm****Longest common substring**

Do two student essays share a common phrase consisting of a sequence of more than 100 letters?

Given the airports at which an airline operates and distances between those airports, what is the shortest total flight distance between two airports?

Given a sorted list of a company's employee records and an employee's first and last name, what is a specific employee's phone number?

Reset

Efficient algorithms and hard problems

Computer scientists and programmers typically focus on using and designing efficient algorithms to solve problems. Algorithm efficiency is most commonly measured by the algorithm runtime, and an efficient algorithm is one whose runtime increases no more than polynomially with respect to the input size. However, some problems exist for which an efficient algorithm is unknown.

NP-complete problems are a set of problems for which no known efficient algorithm exists.¹ NP-⁴⁶² complete problems have the following characteristics:

Taylor Larrechea
COLORADOCSPB2270Summer2023

- No efficient algorithm has been found to solve an NP-complete problem.
- No one has proven that an efficient algorithm to solve an NP-complete problem is impossible.
- If an efficient algorithm exists for one NP-complete problem, then all NP-complete problems can be solved efficiently.

By knowing a problem is NP-complete, instead of trying to find an efficient algorithm to solve the problem, a programmer can focus on finding an algorithm to efficiently find a good, but non-optimal, solution.

PARTICIPATION ACTIVITY

3.1.4: Example NP-complete problem: Cliques.



Animation captions:

1. A programmer may be asked to write an algorithm to solve the problem of determining if a set of K people who all know each other exists within a graph of a social network?
2. For the example social network graph and K = 3, the algorithm should return yes. Xiao, Sean, and Tanya all know each other. Sean, Tanya, and Eve also all know each other.
3. For K = 4, no set of 4 individual who all know each other exists, and the algorithm, should return no.
4. This problem is equivalent to the clique decision problem, which is NP-complete, and no known polynomial time algorithm exists.

PARTICIPATION ACTIVITY

3.1.5: Efficient algorithm and hard problems.



- 1) An algorithm with a polynomial runtime is considered efficient.

©zyBooks 05/27/23 22:12 169246
Taylor Larrechea
COLORADOCSPB2270Summer2023

- True
 False

- 2) An efficient algorithm exists for all computational problems.



- True

False

- 3) An efficient algorithm to solve an NP-complete problem may exist.

 True False

©zyBooks 05/27/23 22:12 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

3.2 Relation between data structures and algorithms

Algorithms for data structures

Data structures not only define how data is organized and stored, but also the operations performed on the data structure. While common operations include inserting, removing, and searching for data, the algorithms to implement those operations are typically specific to each data structure. Ex: Appending an item to a linked list requires a different algorithm than appending an item to an array.

PARTICIPATION ACTIVITY

3.2.1: A list avoids the shifting problem.



Animation content:

undefined

Animation captions:

1. The algorithm to append an item to an array determines the current size, increases the array size by 1, and assigns the new item as the last array element.
2. The algorithm to append an item to a linked list points the tail node's next pointer and the list's tail pointer to the new node.

PARTICIPATION ACTIVITY

3.2.2: Algorithms for data structures.

©zyBooks 05/27/23 22:12 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



Consider the array and linked list in the animation above. Can the following algorithms be implemented with the same code for both an array and linked list?

- 1) Append an item



Yes No2) Return the first item Yes No3) Return the current size Yes No

©zyBooks 05/27/23 22:12 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Algorithms using data structures

Some algorithms utilize data structures to store and organize data during the algorithm execution. Ex: An algorithm that determines a list of the top five salespersons, may use an array to store salespersons sorted by their total sales.

Figure 3.2.1: Algorithm to determine the top five salespersons using an array.

©zyBooks 05/27/23 22:12 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

```
DisplayTopFiveSalespersons(allSalespersons) {
    // topSales array has 5 elements
    // Array elements have subitems for name and total sales
    // Array will be sorted from highest total sales to lowest total sales
    topSales = Create array with 5 elements

    // Initialize all array elements with a negative sales total
    for (i = 0; i < topSales.length; ++i) {
        topSales[i].name = ""
        topSales[i].salesTotal = -1
    }

    for each salesPerson in allSalespersons {
        // If salesPerson's total sales is greater than the last
        // topSales element, salesPerson is one of the top five so far
        if (salesPerson.salesTotal > topSales[topSales.length - 1].salesTotal) {

            // Assign the last element in topSales with the current
            salesperson
            topSales[topSales.length - 1].name = salesPerson.name
            topSales[topSales.length - 1].salesTotal =
            salesPerson.salesTotal

            // Sort topSales in descending order
            SortDescending(topSales)
        }
    }

    // Display the top five salespersons
    for (i = 0; i < topSales.length; ++i) {
        Display topSales[i]
    }
}
```

©zyBooks 05/27/23 22:12 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY**3.2.3: Top five salespersons.**

- 1) Which of the following is *not* equal to the number of items in the topSales array?

- topSales.length
- 5
- allSalesperson.length

©zyBooks 05/27/23 22:12 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 2) To adapt the algorithm to display the top 10 salespersons, what modifications are required?

- Only the array creation



- All loops in the algorithm
 - Both the creation and all loops
- 3) If `allSalespersons` only contains three elements, the `DisplayTopFiveSalespersons` algorithm will display two elements with no name and -1 sales.
- True
 - False

©zyBooks 05/27/23 22:12 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

3.3 Algorithm efficiency

Algorithm efficiency

An algorithm describes the method to solve a computational problem. Programmers and computer scientists should use or write efficient algorithms. **Algorithm efficiency** is typically measured by the algorithm's computational complexity. **Computational complexity** is the amount of resources used by the algorithm. The most common resources considered are the runtime and memory usage.

PARTICIPATION ACTIVITY

3.3.1: Computational complexity.

Animation captions:

1. An algorithm's computational complexity includes runtime and memory usage.
2. Measuring runtime and memory usage allows different algorithms to be compared.
3. Complexity analysis is used to identify and avoid using algorithms with long runtimes or high memory usage.

PARTICIPATION ACTIVITY

3.3.2: Algorithm efficiency and computational complexity.

©zyBooks 05/27/23 22:12 1692462
Taylor Larrechea

COLORADOCSPB2270Summer2023

- 1) Computational complexity analysis allows the efficiency of algorithms to be compared.

- True
- False



2) Two different algorithms that produce the same result have the same computational complexity.

- True
- False

3) Runtime and memory usage are the only two resources making up computational complexity.

- True
- False

©zyBooks 05/27/23 22:12 169246
Taylor Larrechea
COLORADOCSPB2270Summer2023

Runtime complexity, best case, and worst case

An algorithm's **runtime complexity** is a function, $T(N)$, that represents the number of constant time operations performed by the algorithm on an input of size N . Runtime complexity is discussed in more detail elsewhere.

Because an algorithm's runtime may vary significantly based on the input data, a common approach is to identify best and worst case scenarios. An algorithm's **best case** is the scenario where the algorithm does the minimum possible number of operations. An algorithm's **worst case** is the scenario where the algorithm does the maximum possible number of operations.

Input data size must remain a variable

A best case or worst case scenario describes contents of the algorithm's input data only. The input data size must remain a variable, N . Otherwise, the overwhelming majority of algorithms would have a best case of $N=0$, since no input data would be processed. In both theory and practice, saying "the best case is when the algorithm doesn't process any data" is not useful. Complexity analysis always treats the input data size as a variable.

©zyBooks 05/27/23 22:12 169246
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

3.3.3: Linear search best and worst cases.



Animation captions:

1. LinearSearch searches through array elements until finding the key. Searching for 26 requires iterating through the first 3 elements.

2. The search for 26 is neither the best nor the worst case.
3. Searching for 54 only requires one comparison and is the best case: The key is found at the start of the array. No other search could perform fewer operations.
4. Searching for 82 compares against all array items and is the worst case: The number is not found in the array. No other search could perform more operations.

PARTICIPATION ACTIVITY

3.3.4: FindFirstLessThan algorithm best and worst case.

©zyBooks 05/27/23 22:12 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



Consider the following function that returns the first value in a list that is less than the specified value. If no list items are less than the specified value, the specified value is returned.

```
FindFirstLessThan(list, listSize, value) {
    for (i = 0; i < listSize; i++) {
        if (list[i] < value)
            return list[i]
    }
    return value // no lesser value found
}
```

If unable to drag and drop, refresh the page.

Best case**Neither best nor worst case****Worst case**

No items in the list are less than value.

The first half of the list has elements greater than value and the second half has elements less than value.

The first item in the list is less than value.

Reset

©zyBooks 05/27/23 22:12 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

**PARTICIPATION ACTIVITY**

3.3.5: Best and worst case concepts.

- 1) The linear search algorithm's best case scenario is when N = 0.

 True


False

- 2) An algorithm's best and worst case scenarios are always different.

 True False

©zyBooks 05/27/23 22:12 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Space complexity

An algorithm's **space complexity** is a function, $S(N)$, that represents the number of fixed-size memory units used by the algorithm for an input of size N . Ex: The space complexity of an algorithm that duplicates a list of numbers is $S(N) = 2N + k$, where k is a constant representing memory used for things like the loop counter and list pointers.

Space complexity includes the input data and additional memory allocated by the algorithm. An algorithm's **auxiliary space complexity** is the space complexity not including the input data. Ex: An algorithm to find the maximum number in a list will have a space complexity of $S(N) = N + k$, but an auxiliary space complexity of $S(N) = k$, where k is a constant.

PARTICIPATION ACTIVITY

3.3.6: FindMax space complexity and auxiliary space complexity.



Animation content:

undefined

Animation captions:

1. FindMax's arguments represent input data. Non-input data includes variables allocated in the function body: maximum and i.
2. The list's size is a variable, N . Three integers are also used, listSize, maximum, and i, making the space complexity $S(N) = N + 3$.
3. The auxiliary space complexity includes only the non-input data, which does not increase for larger input lists.
4. The function's auxiliary space complexity is $S(N) = 2$.

©zyBooks 05/27/23 22:12 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

3.3.7: Space complexity of GetEvens function.



Consider the following function, which builds and returns a list of even numbers from the input list.

```
GetEvens(list, listSize) {
    i = 0
    evensList = Create new, empty list
    while (i < listSize) {
        if (list[i] % 2 == 0)
            Add list[i] to evensList
        i = i + 1
    }
    return evensList
}
```

©zyBooks 05/27/23 22:12 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

- 1) What is the maximum possible size of the returned list?

listSize
 listSize / 2

- 2) What is the minimum possible size of the returned list?

listSize / 2
 1
 0

- 3) What is the worst case auxiliary space complexity of GetEvens if N is the list's size and k is a constant?

$S(N) = N + k$
 $S(N) = k$

- 4) What is the best case auxiliary space complexity of GetEvens if N is the list's size and k is a constant?

$S(N) = N + k$
 $S(N) = k$



©zyBooks 05/27/23 22:12 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

4.1 Why pointers?

A challenging and yet powerful programming construct is something called a *pointer*. A **pointer** is a variable that contains a memory address. This section describes a few situations where pointers are useful.

Vectors use dynamically allocated arrays

The C++ vector class is a container that internally uses a **dynamically allocated array**, an array whose size can change during runtime. When a vector is created, the vector class internally dynamically allocates an array with an initial size, such as the size specified in the constructor. If the number of elements added to the vector exceeds the capacity of the current internal array, the vector class will dynamically allocate a new array with an increased size, and the contents of the array are copied into the new larger array. Each time the internal array is dynamically allocated, the array's location in memory will change. Thus, the vector class uses a pointer variable to store the memory location of the internal array.

The ability to dynamically change the size of a vector makes vectors more powerful than arrays. Built-in constructs have also made vectors safer to use in terms of memory management.

PARTICIPATION ACTIVITY

4.1.1: Dynamically allocated arrays.



Animation content:

Animation captions:

1. A vector internally uses a dynamically allocated array, an array whose size can change at runtime. To create a dynamically allocated array, a pointer stores the memory location of the array.
2. A vector internally has data members for the size and capacity. Size is the current number of elements in the vector. capacity is the maximum number of elements that can be stored in the allocated array.
3. push_back(2) needs to add a 6th element to the vector, but the capacity is only 5. push_back() allocates a new array with a larger capacity, copies existing elements to the new array, and adds the new element.
4. Internally, the pointer for the vector's internal array is assigned to point to the new array, capacity is assigned with the new maximum size, size is incremented, and the previous array is freed.

PARTICIPATION ACTIVITY

4.1.2: Dynamically allocated arrays.



- 1) The size of a vector is the same as the vector's capacity.

- True
- False

- 2) When a dynamically allocated array increases capacity, the array's memory location remains the same.

- True

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

False

- 3) Data that is stored in memory and no longer being used should be deleted to free up the memory.

 True False

@zyBooks 05/30/23 21:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Inserting/erasing in vectors vs. linked lists

A vector (or array) stores a list of items in contiguous memory locations, which enables immediate access to any element of a vector `userGrades` by using `userGrades.at(i)` (or `userGrades[i]`). However, inserting an item requires making room by shifting higher-indexed items. Similarly, erasing an item requires shifting higher-indexed items to fill the gap. Shifting each item requires a few operations. If a program has a vector with thousands of elements, a single call to `insert()` or `erase()` can require thousands of instructions and cause the program to run very slowly, often called the **vector insert/erase performance problem**.

PARTICIPATION ACTIVITY

4.1.3: Vector insert performance problem.



Animation content:

undefined

Animation captions:

1. Inserting an item at a specific location in a vector requires making room for the item by shifting higher-indexed items.

A programmer can use a linked list to make inserts or erases faster. A **linked list** consists of items that contain both data and a pointer—a *link*—to the next list item. Thus, inserting a new item B between existing items A and C just requires changing A to point to B's memory location, and B to point to C's location, as shown in the following animation. No shifts occur.

PARTICIPATION ACTIVITY

4.1.4: A list avoids the shifting problem.



Animation content:

undefined

Animation captions:

1. List's first two items initially: (A, C, ...). Item A points to the next item at location 88. Item C points to next item at location 113 (not shown).
2. To insert new item B after item A, the new item B is first added to memory at location 90.
3. Item B is set to point to location 88. Item A is updated to point to location 90. New list is (A, B, C, ...). No shifting of items after C was required.

@zyBooks 05/30/23 21:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

A vector is like people ordered by their seat in a theater row; if you want to insert yourself between two adjacent people, other people have to shift over to make room. A linked list is like people ordered by holding hands; if you want to insert yourself between two people, only those two people have to change hands, and nobody else is affected.

Table 4.1.1: Comparing vectors and linked lists.

Vector	Linked list
<ul style="list-style-type: none"> Stores items in contiguous memory locations Supports quick access to i^{th} element via <code>at(i)</code> <ul style="list-style-type: none"> May be slow for inserts or erases on large lists due to necessary shifting of elements 	<ul style="list-style-type: none"> Stores each item anywhere in memory, with each item pointing to the next list item Supports fast inserts or deletes <ul style="list-style-type: none"> access to i^{th} element may be slow as the list must be traversed from the first item to the i^{th} item Uses more memory due to storing a link for each item

PARTICIPATION ACTIVITY

4.1.5: Inserting/erasing in vectors vs. linked lists.



For each operation, how many elements must be shifted? Assume no new memory needs to be allocated. Questions are for vectors, but also apply to arrays.

- 1) Append an item to the end of a 999-element vector (e.g., using `push_back()`).

Check**Show answer**

- 2) Insert an item at the front of a 999-element vector.

Check**Show answer**

- 3) Delete an item from the end of a 999-element vector.

Check**Show answer**

- 4) Delete an item from the front of a 999-element vector.

Check**Show answer**

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



- 5) Appending an item at the end of a 999-item linked list.



Check**Show answer**

- 6) Inserting a new item between the 10th and 11th items of a 999-item linked list.

Check**Show answer**

©zyBooks 05/30/23 21:54 1692462

Taylor Larrechea

COLORADO CSPB2270Summer2023



- 7) Finding the 500th item in a 999-item linked list requires visiting how many items? Correct answer is one of 0, 1, 500, and 999.

Check**Show answer**

Pointers used to call class member functions

When a class member function is called on an object, a pointer to the object is automatically passed to the member function as an implicit parameter called the **this** pointer. The **this** pointer enables access to an object's data members within the object's class member functions. A data member can be accessed using **this** and the member access operator for a pointer, `->`, ex. `this->sideLength`. The **this** pointer clearly indicates that an object's data member is being accessed, which is needed if a member function's parameter has the same variable name as the data member. The concept of the **this** pointer is explained further elsewhere.

PARTICIPATION
ACTIVITY

4.1.6: Pointers used to call class member functions.



Animation content:

undefined

Animation captions:

1. square1 is a ShapeSquare object that has a double sideLength data member and a SetSideLength() member function.
2. Member functions have an implicit 'this' implicit parameter, which is a pointer to the class type. SetSideLength()'s implicit this parameter is a pointer to a ShapeSquare object.
3. When square1's SetSideLength() member function is called, square1's memory address is passed to the function using the 'this' implicit parameter.
4. The implicitly-passed square1 object pointer is clearly accessed within the member function via the name "this".

PARTICIPATION
ACTIVITY

4.1.7: The 'this' pointer.



Assume the class FilmInfo has a private data member int filmLength and a member function
void SetFilmLength(int filmLength).

1) In SetFilmLength(), which would assign the data member filmLength with the value 120?

- this->filmLength = 120;
- this.filmLength = 120;
- 120 = this->filmLength;

2) In SetFilmLength(), which would assign the data member filmLength with the parameter filmLength?

- filmLength = filmLength;
- this.filmLength = filmLength;
- this->filmLength = filmLength;

©zyBooks 05/30/23 21:54 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

Exploring further:

- [Pointers tutorial](#) from cplusplus.com
- [Pointers article](#) from cplusplus.com

4.2 Pointer basics

Pointer variables

A **pointer** is a variable that holds a memory address, rather than holding data like most variables. A pointer has a data type, and the data type determines what type of address is held in the pointer. Ex: An integer pointer holds a memory address of an integer, and a double pointer holds an address of a double. A pointer is declared by including * before the pointer's name. Ex: **int* maxItemPointer** declares an integer pointer named maxItemPointer.

Typically, a pointer is initialized with another variable's address. The **reference operator** (&) obtains a variable's address. Ex: **&someVar** returns the memory address of variable someVar. When a pointer is initialized with another variable's address, the pointer "points to" the variable.

PARTICIPATION ACTIVITY

4.2.1: Assigning a pointer with an address.



©zyBooks 05/30/23 21:54 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

Animation content:

undefined

Animation captions:

1. someInt is located in memory at address 76.
2. valPointer is located in memory at address 78. valPointer has not been initialized, so valPointer points to an unknown address.
3. someInt is assigned with 5. The reference operator & returns someInt's address 76.

4. valPointer is assigned with the memory address of someInt, so valPointer points to someInt.

Printing memory addresses

The examples in this material show memory addresses using decimal numbers for simplicity. Outputting a memory address is likely to display a hexadecimal value like 006FF978 or 0x7ffc3ae4f0e4. Hexadecimal numbers are base 16, so the values use the digits 0-9 and letters A-F.

zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

4.2.2: Declaring and initializing a pointer.



- 1) Declare a double pointer called sensorPointer.

Check

[Show answer](#)

- 2) Output the address of the double variable sensorVal.

`cout <<` `;`

Check

[Show answer](#)

- 3) Assign sensorPointer with the variable sensorVal's address. In other words, make sensorPointer point to sensorVal.

Check

[Show answer](#)

Dereferencing a pointer

The **dereference operator** (*) is prepended to a pointer variable's name to retrieve the data to which the pointer variable points. Ex: If valPointer points to a memory address containing the integer 123, then `cout << *valPointer;` dereferences valPointer and outputs 123.

zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

4.2.3: Using the dereference operator.



Animation content:

undefined

Animation captions:

1. somelnt is located in memory at address 76, and valPointer points to somelnt.
2. The dereference operator * gets the value pointed to by valPointer, which is 5.
3. Assigning *valPointer with a new value changes the value valPointer points to. The 5 changes to 10.
4. Changing *valPointer also changes somelnt. somelnt is now 10.

PARTICIPATION ACTIVITY**4.2.4: Dereferencing a pointer.**

@zyBooks 05/30/23 21:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Refer to the code below.

```
char userLetter = 'B';
char* letterPointer;
```

1) What line of code makes letterPointer

point to userLetter?



- letterPointer =

 userLetter;
- *letterPointer =

 &userLetter;
- letterPointer =

 &userLetter;

2) What line of code assigns the variable

outputLetter with the value letterPointer

points to?



- outputLetter =

 letterPointer;
- outputLetter =

 *letterPointer;
- someChar =

 &letterPointer;

3) What does the code output?



```
letterPointer = &userLetter;
userLetter = 'A';
*letterPointer = 'C';
cout << userLetter;
```

- A
- B
- C

@zyBooks 05/30/23 21:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Null pointer

When a pointer is declared, the pointer variable holds an unknown address until the pointer is initialized. A programmer may wish to indicate that a pointer points to "nothing" by initializing a pointer to null. **Null** means "nothing". A pointer that is assigned with the keyword **nullptr** is said to be null. Ex: `int *maxValPointer = nullptr;` makes maxValPointer null.

In the animation below, the function PrintValue() only outputs the value pointed to by valuePointer if valuePointer is not null.

PARTICIPATION ACTIVITY**4.2.5: Checking to see if a pointer is null.**

Animation content:

undefined

Animation captions:

1. someInt is located in memory at address 76. valPointer is assigned nullptr, so valPointer is null.
2. valPointer is passed to PrintValue(), so the valuePointer parameter is assigned nullptr.
3. The if statement is true since valuePointer is null.
4. valPointer points to someInt, so calling PrintValue() assigns valuePointer with the address 76.
5. The if statement is false because valuePointer is no longer null. valuePointer points to the value 5, so 5 is output.

Null pointer

The nullptr keyword was added to the C++ language in version C++11. Before C++11, common practice was to use the literal 0 to indicate a null pointer. In C++'s predecessor language C, the macro NULL is used to indicate a null pointer.

PARTICIPATION ACTIVITY

4.2.6: Null pointer.



Refer to the animation above.

- 1) The code below outputs 3.



```
int numSides = 3;
int* valPointer = &numSides;
PrintValue(valPointer);
```

- True
- False

- 2) The code below outputs 5.



```
int numSides = 5;
int* valPointer = &numSides;
valPointer = nullptr;
PrintValue(valPointer);
```

- True
- False

- 3) The code below outputs 7.

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

```
int numSides = 7;
int* valPointer = nullptr;
cout << *valPointer;
```

- True
- False

Common pointer errors

A number of common pointer errors result in syntax errors that are caught by the compiler or runtime errors that may result in the program crashing.

Common syntax errors:

- A common error is to use the dereference operator when initializing a pointer. Ex: For a variable declared `int maxValue;` and a pointer declared `int* valPointer;`, `*valPointer = &maxValue;` is a syntax error because `*valPointer` is referring to the value pointed to, not the pointer itself.
- A common error when declaring multiple pointers on the same line is to forget the `*` before each pointer name. Ex: `int* valPointer1, valPointer2;` declares `valPointer1` as a pointer, but `valPointer2` is declared as an integer because no `*` exists before `valPointer2`. Good practice is to declare one pointer per line to avoid accidentally declaring a pointer incorrectly.

©zyBooks 05/30/23 21:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Common runtime errors:

- A common error is to use the dereference operator when a pointer has not been initialized. Ex: `cout << *valPointer;` may cause a program to crash if `valPointer` holds an unknown address or an address the program is not allowed to access.
- A common error is to dereference a null pointer. Ex: If `valPointer` is null, then `cout << *valPointer;` causes the program to crash. A pointer should always hold a valid address before the pointer is dereferenced.

PARTICIPATION ACTIVITY

4.2.7: Common pointer errors.



Animation content:

undefined

Animation captions:

1. A syntax error results if `valPointer` is assigned using the dereference operator `*`.
2. Multiple pointers cannot be declared on a single line with only one asterisk. Good practice is to declare each pointer on a separate line.
3. `valPointer` is not initialized, so `valPointer` contains an unknown address. Dereferencing an unknown address may cause a runtime error.
4. `valPointer` is null, and dereferencing a null pointer causes a runtime error.

PARTICIPATION ACTIVITY

4.2.8: Common pointer errors.



Indicate if each code segment has a syntax error, runtime error, or no error.

1)

```
char* newPointer;
*newPointer = 'A';
cout << *newPointer;
```

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- syntax error
- runtime error
- no errors

2)



```
char* valPointer1, *valPointer2;
valPointer1 = nullptr;
valPointer2 = nullptr;
```

- syntax error
- runtime error
- no errors

3)

```
char someChar = 'z';
char* valPointer;
*valPointer = &someChar;
```

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- syntax error
- runtime error
- no errors

4)

```
char* newPointer = nullptr;
char someChar = 'A';
*newPointer = 'B';
```

- syntax error
- runtime error
- no errors

Two pointer declaration styles

Some programmers prefer to place the asterisk next to the variable name when declaring a pointer. Ex: `int *valPointer;`. The style preference is useful when declaring multiple pointers on the same line: `int *valPointer1, *valPointer2;`. Good practice is to use the same pointer declaration style throughout the code:
Either `int* valPointer` or `int *valPointer`.

This material uses the style `int* valPointer` and always declares one pointer per line to avoid accidentally declaring a pointer incorrectly.

Advanced compilers can check for common errors

Some compilers have advanced code analysis capabilities to catch some runtime errors at compile time. Ex: The compiler may issue a warning if the compiler detects a null pointer is being dereferenced. An advanced compiler can never catch all runtime errors because a potential runtime error may depend on user input, which is unknown at compile time.

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

The following provides an example (not useful other than for learning) of assigning the address of variable vehicleMpg to the pointer variable valPointer.

1. Run and observe that the two output statements produce the same output.
2. Modify the value assigned to *valPointer and run again.
3. Now uncomment the statement that assigns vehicleMpg. PREDICT whether both output statements will print the same output. Then run and observe the output. Did you predict correctly?

Load default template...
Run

```

1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     double vehicleMpg;
6     double* valPointer = nullptr;
7
8     valPointer = &vehicleMpg;
9
10    *valPointer = 29.6; // Assigns the value
11                                // POINTED TO
12
13    // vehicleMpg = 40.0; // Uncomment this line
14
15    cout << "Vehicle MPG = " << vehicleMpg;
16
17 }
```

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

CHALLENGE ACTIVITY

4.2.1: Enter the output of pointer content.



489394.3384924.qx3zqy7

Start

Type the program's output

```
#include <iostream>
using namespace std;

int main() {
    int someNumber;
    int* numberPointer;

    someNumber = 10;
    numberPointer = &someNumber;

    cout << someNumber << " " << *numberPointer << endl;

    return 0;
}
```



1

2

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Check

Next

CHALLENGE ACTIVITY

4.2.2: Printing with pointers.



If the input is negative, make numItemsPointer be null. Otherwise, make numItemsPointer point to numItems and multiply the value to which numItemsPointer points by 10. Ex: If the user enters 99, the output should be:

Items: 990

[Learn how our autograder works](#)

489394.3384924.qx3zqy7

©zyBooks 05/30/23 21:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

```

1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     int* numItemsPointer;
6     int numItems;
7
8     cin >> numItems;
9
10    /* Your solution goes here */
11
12    if (numItemsPointer == nullptr) {
13        cout << "Items is negative" << endl;
14    }
15    else {
16        cout << "Items is positive" << endl;
17    }
18}
```

Run

View your last submission ▾

**CHALLENGE
ACTIVITY**

4.2.3: Pointer basics.



489394.3384924.qx3zqy7

Start

Given variables level, time, and alert, declare and assign the following pointers:

- double pointer levelPointer is assigned with the address of level.
- integer pointer timePointer is assigned with the address of time.
- character pointer alertPointer is assigned with the address of alert.

Ex: If the input is 7.5 19 H, then the output is:

Tide level: 7.5 meters
Recorded at hour: 19
Alert: H

©zyBooks 05/30/23 21:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

```

1 #include <iostream>
2 #include <iomanip>
3 using namespace std;
4
5 int main() {
6     double level;
7     int time;
8     char alert;
9 }
```

```

10  /* Your code goes here */
11
12  cin >> level;
13  cin >> time;
14  cin >> alert;
15

```

1

2

©zyBooks 05/30/23 21:54 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

Check**Next level**

4.3 Operators: new, delete, and ->

The new operator

The **new operator** allocates memory for the given type and returns a pointer to the allocated memory. If the type is a class, the new operator calls the class's constructor after allocating memory for the class's member variables.

PARTICIPATION ACTIVITY

4.3.1: The new operator allocates space for an object, then calls the constructor.



Animation content:

undefined

Animation captions:

1. The Point class contains two members, X and Y, both doubles.
2. The new operator does 2 things. First, enough space is allocated for the Point object's 2 members, starting at memory address 46.
3. Then the Point constructor is called, displaying a message and setting the X and Y values.
4. The new operator returns a pointer to the allocated and initialized memory at address 46.

PARTICIPATION ACTIVITY

4.3.2: The new operator.



- 1) The new operator returns an int.

- True
- False

©zyBooks 05/30/23 21:54 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023



- 2) When used with a class type, the new operator allocates memory after calling the class's constructor.

- True
- False



- 3) The new operator allocates, but does not deallocate, memory.

- True
- False

Constructor arguments

The new operator can pass arguments to the constructor. The arguments must be in parentheses following the class name.

@zyBooks 5/30/23 9:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION
ACTIVITY

4.3.3: Constructor arguments.



Animation content:

undefined

Animation captions:

1. The Point class contains 2 doubles, X and Y. The constructor has 2 parameters.
2. "new Point" calls the constructor with no arguments. The default value of 0 is used for both numbers.
3. point1 is a pointer to the allocated object that resides at address 60. point1 is dereferenced, and the Print() member function is called.
4. "new Point(8, 9)" passes 8 and 9 as the constructor arguments.
5. point2 points to the object at address 63. Print() is called for point2.

PARTICIPATION
ACTIVITY

4.3.4: Constructor arguments.



If unable to drag and drop, refresh the page.

`Point* point = new Point(0, 10);`

`Point* point = new Point(0, 0, 0);`

`Point* point = new Point(10);`

`Point* point = new Point();`

Constructs the point (0, 10).

Constructs the point (0, 0).

Constructs the point (10, 0).

Constructs the point (0, 10).

Causes a compiler error.

@zyBooks 05/30/23 21:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Reset

The member access operator

When using a pointer to an object, the **member access operator** (`->`) allows access to the object's members with the syntax `a->b` instead of `(*a).b`. Ex: If `myPoint` is a pointer to a `Point` object, `myPoint->Print()` calls the `Print()` member function.

Table 4.3.1: Using the member access operator.

Action	Syntax with dereferencing	Syntax with member access operator
Display point1's Y member value with cout	<code>cout << (*point1).Y;</code>	<code>cout << point1->Y;</code>
Call point2's Print() member function	<code>(*point2).Print();</code>	<code>point2->Print();</code>

PARTICIPATION ACTIVITY

4.3.5: The member access operator.



- 1) Which statement calls point1's `Print()` member function?

```
Point point1(20, 30);

 (*point1).Print();
 point1->Print();
 point1.Print();
```

- 2) Which statement calls point2's `Print()` member function?

```
Point* point2 = new Point(16,
8);

 point2.Print();
 point2->Print();
```

- 3) Which statement is *not* valid for multiplying point3's X and Y members?

```
Point* point3 = new Point(100,
50);

 point3->X * point3->Y
 point3->X * (*point3).Y
 point3->X (*point3).Y
```

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

The delete operator

The **delete operator** deallocates (or frees) a block of memory that was allocated with the `new` operator. The statement `delete pointerVariable;` deallocates a memory block pointed to by `pointerVariable`. If `pointerVariable` is null, `delete` has no effect.

After the `delete`, the program should not attempt to dereference `pointerVariable` since `pointerVariable` points to a memory location that is no longer allocated for use by `pointerVariable`. *Dereferencing a pointer whose memory has been deallocated is a common error and may cause strange program behavior that is difficult to debug. Ex: If `pointerVariable` points to deallocated*

memory that is later allocated to someVariable, changing *pointerVariable will mysteriously change someVariable. Calling delete with a pointer that wasn't previously set by the new operator has undefined behavior and is a logic error.

**PARTICIPATION
ACTIVITY**

4.3.6: The delete operator.


Animation content:

undefined

©zyBooks 05/30/23 21:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Animation captions:

1. point1 is allocated, and the X and Y members are displayed.
2. Deleting point1 frees the memory for the X and Y members. point1 still points to address 87.
3. Since point1 points to deallocated memory, attempting to use point1 after deletion is a logic error.

**PARTICIPATION
ACTIVITY**

4.3.7: The delete operator.



- 1) The delete operator can affect any pointer.
 - True
 - False
- 2) The statement `delete point1;` throws an exception if point1 is null.
 - True
 - False
- 3) After the statement `delete point1;` executes, point1 will be null.
 - True
 - False

Allocating and deleting object arrays

The new operator creates a dynamically allocated array of objects if the class name is followed by square brackets containing the array's length. A single, contiguous chunk of memory is allocated for the array, then the default constructor is called for each object in the array. A compiler error occurs if the class does not have a constructor that can take 0 arguments.

The **delete[] operator** is used to free an array allocated with the new operator.

**PARTICIPATION
ACTIVITY**

4.3.8: Allocating and deleting an array of Point objects.

©zyBooks 05/30/23 21:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Animation content:

undefined

Animation captions:

1. The new operator allocates a contiguous chunk of memory for an array of 4 Point objects. The default constructor is called for each, setting X and Y to 0.
2. Each point in the array is displayed.
3. The entire array is freed with the delete[] operator.

PARTICIPATION ACTIVITY

4.3.9: Allocating and deleting object arrays.



- 1) The array of points from the example above ____ contiguous in memory.

- might or might not be
- is always

- 2) What code properly frees the dynamically allocated array below?

```
Airplane* airplanes = new  
Airplane[10];  
  
 delete airplanes;  
 delete[] airplanes;  
 for (int i = 0; i < 10;  
     ++i) {  
    delete airplanes[i];  
}
```



- 3) The statement below only works if the Dalmatian class has ____.

```
Dalmatian* dogs = new  
Dalmatian[101];  
  
 no member functions  
 only numerical member variables  
 a constructor that can take 0 arguments
```

**CHALLENGE ACTIVITY**

4.3.1: Using the new, delete, and -> operators.



489394.3384924.qx3zqy7

Start

Type the program's output

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

```
#include <iostream>
using namespace std;

class Car {
public:
    Car(int distanceToSet);
private:
    int distanceTraveled;
};

Car::Car(int distanceToSet) {
    distanceTraveled = distanceToSet;
    cout << "Traveled: " << distanceTraveled << endl;
}

int main() {
    Car* myCar1 = nullptr;
    Car* myCar2 = nullptr;

    myCar1 = new Car(55);
    myCar2 = new Car(60);

    return 0;
}
```

@zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

1

2

Check**Next****CHALLENGE ACTIVITY**

4.3.2: Deallocating memory



Deallocate memory for kitchenPaint using the delete operator. Note: Destructors, which use the "~" character, are explained in a later section.

[Learn how our autograder works](#)

489394.3384924.qx3zqy7

```
1 #include <iostream>
2 using namespace std;
3
4 class PaintContainer {
5 public:
6     ~PaintContainer();
7     double gallonPaint;
8 };
9
10 PaintContainer::~PaintContainer() { // Covered in section on Destructors.
11     cout << "PaintContainer deallocated." << endl;
12 }
13
14 int main() {
15     PaintContainer* kitchenPaint;
```

@zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Run

View your last submission ▾

CHALLENGE ACTIVITY

4.3.3: Operators: new, delete, and ->.



489394.3384924.qx3zqy7

Start

Two integers are read as the velocity and the duration of a MovingBody object. Assign pointer myMovingBody object using the velocity and the duration as arguments in that order.

Ex: If the input is 8 13, then the output is:

```
MovingBody's velocity: 8
MovingBody's duration: 13
```

```
1 #include <iostream>
2 using namespace std;
3
4 class MovingBody {
5 public:
6     MovingBody(int velocityValue, int durationValue);
7     void Print();
8 private:
9     int velocity;
10    int duration;
11 };
12 MovingBody::MovingBody(int velocityValue, int durationValue) {
13     velocity = velocityValue;
14     duration = durationValue;
15 }
```

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

1

2

3

4

Check**Next level**

Exploring further:

- [operator new\[\] Reference Page](#) from cplusplus.com
- [More on operator new\[\]](#) from msdn.microsoft.com
- [operator delete\[\] Reference Page](#) from cplusplus.com
- [More on delete operator](#) from msdn.microsoft.com
- [More on -> operator](#) from msdn.microsoft.com

4.4 String functions with pointers

C string library functions

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

The C string library, introduced elsewhere, contains several functions for working with C strings. This section describes the use of char pointers in such functions. The C string library must first be included via: `#include <cstring>`.

Each string library function operates on one or more strings, each passed as a `char*` or `const char*` argument. Strings passed as `char*` can be modified by the function, whereas strings passed as `const char*` arguments cannot. Examples of such functions are `strcmp()` and `strcpy()`, introduced elsewhere.



4.4.1: strcmp() and strcpy() string functions.

Animation content:

undefined

Animation captions:

1. strcmp() compares 2 strings. Since neither string is modified during the comparison, each parameter is a const char*.
2. strcmp() returns an integer that is 0 if the strings are equal, non-zero if the strings are not equal.
3. strcpy() copies a source string to a destination string. The destination string gets modified and thus is a char*.
4. The source string is not modified and thus is a const char*.
5. strcpy() copies 4 characters, "xyz" and the null-terminator, to newText.

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

4.4.2: C string library functions.



- 1) A variable declared as `char*`

`substringAt5 = &myString[5];`
cannot be passed as an argument to strcmp(), since strcmp() requires const char* arguments.



- True
 False

- 2) A character array variable declared as

`char myString[50];` can be passed as either argument to strcpy().



- True
 False

- 3) A variable declared as `const char*`

`firstMonth = "January";` could be passed as either argument to strcpy().



- True
 False

C string search functions

©zyBooks 05/30/23 21:54 1692462

strchr(), strrchr(), and strstr() are C string library functions that search strings for an occurrence of a character or substring. Each function's first parameter is a const char*, representing the string to search within.

Taylor Larrechea
COLORADOCSPB2270Summer2023

The strchr() and strrchr() functions find a character within a string, and thus have a char as the second parameter. strchr() finds the first occurrence of the character within the string and strrchr() finds the last occurrence.

strstr() searches for a substring within another string, and thus has a const char* as the second parameter.

Table 4.4.1: Some C string search functions.

Given:

```
char orgName[100] = "The Dept. of Redundancy Dept.";
char newText[100];
char* subString = nullptr;
```

strchr()	<p><code>strchr(sourceStr, searchChar)</code></p> <p>Returns a null pointer if searchChar does not exist in sourceStr. Else, returns pointer to first occurrence.</p>	<pre>if (strchr(orgName, 'D') != nullptr) { // 'D' exists in orgName? subString = strchr(orgName, 'D'); // Points to first 'D' strcpy(newText, subString); // newText now "Dept. of Redundancy Dept." } if (strchr(orgName, 'Z') != nullptr) { // 'Z' exists in orgName? ... // Doesn't exist, branch not taken }</pre>
strrchr()	<p><code>strrchr(sourceStr, searchChar)</code></p> <p>Returns a null pointer if searchChar does not exist in sourceStr. Else, returns pointer to LAST occurrence (searches in reverse, hence middle 'r' in name).</p>	<pre>if (strrchr(orgName, 'D') != nullptr) { // 'D' exists in orgName? subString = strrchr(orgName, 'D'); // Points to last 'D' strcpy(newText, subString); // newText now "Dept." }</pre>
strstr()	<p><code>strstr(str1, str2)</code></p> <p>Returns a null pointer if str2 does not exist in str1. Else, returns a char pointer pointing to the first character of the first occurrence of string str2 within string str1.</p>	<pre>subString = strstr(orgName, "Dept"); // Points to first 'D' if (subString != nullptr) { strcpy(newText, subString); // newText now "Dept. of Redundancy Dept." }</pre>

PARTICIPATION ACTIVITY

4.4.3: C string search functions.



- 1) What does fileExtension point to after the following code?

```
const char* fileName =
"Sample.file.name.txt";
const char* fileExtension =
strrchr(fileName, '.');
```

- ".file.name.txt"
- ".txt"
- "Sample.file.name"

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



2) `strstr(fileName, ".pdf")` is non-null only if the `fileName` string ends with `".pdf"`.

- True
- False

3) What is true about `fileName` if the following expression evaluates to true?

```
strchr(fileName, '.') ==  
strrchr(fileName, '.')
```



©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- The `'.'` character occurs exactly once in `fileName`.
- The `'.'` character occurs 0 or 1 time in `fileName`.
- The `'.'` character occurs 1 or more times in `fileName`.

Search and replace example

The following example carries out a simple censoring program, replacing an exclamation point by a period and "Boo" by "---" (assuming those items are somehow bad and should be censored.) "Boo" is replaced using the `strncpy()` function, which is described elsewhere.

Note that only the first occurrence of "Boo" is replaced, as `strstr()` returns a pointer to the first occurrence. Additional code would be needed to delete all occurrences.

Figure 4.4.1: String searching example.

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

```
#include <iostream>
#include <cstring>
using namespace std;

int main() {
    const int MAX_USER_INPUT = 100;           // Max
    input size
    char userInput[MAX_USER_INPUT];          // User
    defined string
    char* stringPos = nullptr;                // Index
    into string

    // Prompt user for input
    cout << "Enter a line of text: ";
    cin.getline(userInput, MAX_USER_INPUT);

    // Locate exclamation point, replace with period
    stringPos = strchr(userInput, '!');

    if (stringPos != nullptr) {
        *stringPos = '.';
    }

    // Locate "Boo" replace with "---"
    stringPos = strstr(userInput, "Boo");

    if (stringPos != nullptr) {
        strncpy(stringPos, "___", 3);
    }

    // Output modified string
    cout << "Censored: " << userInput << endl;

    return 0;
}
```

@zyBooks 05/30/23 21:54 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

```
Enter a line of text:
Hello!
Censored: Hello.

...
Enter a line of text: Boo
hoo to you!
Censored: ___ hoo to you.

...
Enter a line of text: Booo!
Boooo!!!!
Censored: ___o. Boooo!!!!
```

**PARTICIPATION
ACTIVITY**
4.4.4: Modifying and searching strings.


- 1) Declare a `char*` variable named `charPtr`.

Check
Show answer

- 2) Assuming `char* firstR;` is already declared, store in `firstR` a pointer to the first instance of an 'r' in the `char*` variable `userInput`.

@zyBooks 05/30/23 21:54 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023



- 3) Assuming `char* lastR;` is already declared, store in `lastR` a pointer to the last instance of an 'r' in the `char*` variable `userInput`.



Check**Show answer**

- 4) Assuming `char* firstQuit;` is already declared, store in `firstQuit` a pointer to the first instance of "quit" in the `char*` variable `userInput`.

**Check****Show answer**

©zyBooks 05/30/23 21:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

CHALLENGE ACTIVITY

4.4.1: Enter the output of the string functions.



489394.3384924.qx3zqy7

Start

Type the program's output

```
#include <iostream>
#include <cstring>
using namespace std;

int main() {
    char nameAndTitle[50];
    char* stringPointer = nullptr;

    strcpy(nameAndTitle, "Dr. Ron Smith");

    stringPointer = strchr(nameAndTitle, 'S');
    if (stringPointer != nullptr) {
        cout << "a" << endl;
    }
    else {
        cout << "b" << endl;
    }

    return 0;
}
```



1

2

3

4

Check**Next****CHALLENGE ACTIVITY**

4.4.2: Find char in C string

Assign a pointer to any instance of `searchChar` in `personName` to `searchResult`.

©zyBooks 05/30/23 21:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

489394.3384924.qx3zqy7

```
1 #include <iostream>
2 #include <cstring>
3 using namespace std;
4
5 int main() {
6     char personName[100];
7     char searchChar;
8     char* searchResult = nullptr;
```

```

9
10   cin.getline(personName, 100);
11   cin >> searchChar;
12
13  /* Your solution goes here */
14
15  if (searchResult != nullptr) {
16      cout << "Search result: " << endl;

```

Run

@zyBooks 05/30/23 21:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

View your last submission ▾

**CHALLENGE
ACTIVITY**

4.4.3: Find C string in C string.



Assign the first instance of "The" in movieTitle to movieResult.

[Learn how our autograder works](#)

489394.3384924.qx3zqy7

```

1 #include <iostream>
2 #include <cstring>
3 using namespace std;
4
5 int main() {
6     char movieTitle[100];
7     char* movieResult = nullptr;
8
9     cin.getline(movieTitle, 100);
10
11    /* Your solution goes here */
12
13    cout << "Movie title contains The? ";
14    if (movieResult != nullptr) {
15        cout << "Yes." << endl;

```

Run

View your last submission ▾

4.5 A first linked list

A common use of pointers is to create a list of items such that an item can be efficiently inserted somewhere in the middle of the list, without the shifting of later items as required for a vector. The following program illustrates how such a list can be created. A class is defined to represent each list item, known as a **list node**. A node is comprised of the data to be stored in each list item, in this case just one int, and a pointer to the next node in the list. A special node named head is created to represent the front of the list, after which regular items can be inserted.

Figure 4.5.1: A basic example to introduce linked lists.

-1
555
777
999

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

```

#include <iostream>
using namespace std;

class IntNode {
public:
    IntNode(int dataInit = 0, IntNode* nextLoc =
nullptr);
    void InsertAfter(IntNode* nodeLoc);
    IntNode* GetNext();
    void PrintNodeData();
private:
    int dataVal;
    IntNode* nextNodePtr;
};

// Constructor
IntNode::IntNode(int dataInit, IntNode* nextLoc) {
    this->dataVal = dataInit;
    this->nextNodePtr = nextLoc;
}

/* Insert node after this node.
 * Before: this -- next
 * After:  this -- node -- next
 */
void IntNode::InsertAfter(IntNode* nodeLoc) {
    IntNode* tmpNext = nullptr;

    tmpNext = this->nextNodePtr; // Remember next
    this->nextNodePtr = nodeLoc; // this -- node --
?                                         // this -- node --
nodeLoc->nextNodePtr = tmpNext; // this -- node --
next
}

// Print dataVal
void IntNode::PrintNodeData() {
    cout << this->dataVal << endl;
}

// Grab location pointed by nextNodePtr
IntNode* IntNode::GetNext() {
    return this->nextNodePtr;
}

int main() {
    IntNode* headObj = nullptr; // Create IntNode
objects
    IntNode* nodeObj1 = nullptr;
    IntNode* nodeObj2 = nullptr;
    IntNode* nodeObj3 = nullptr;
    IntNode* currObj = nullptr;

    // Front of nodes list
    headObj = new IntNode(-1);

    // Insert nodes
    nodeObj1 = new IntNode(555);
    headObj->InsertAfter(nodeObj1);

    nodeObj2 = new IntNode(999);
    nodeObj1->InsertAfter(nodeObj2);

    nodeObj3 = new IntNode(777);
    nodeObj2->InsertAfter(nodeObj3);

    // Print linked list
    currObj = headObj;
    while (currObj != nullptr) {
        currObj->PrintNodeData();
        currObj = currObj->GetNext();
    }
}

```

©zyBooks 05/30/23 21:54 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

©zyBooks 05/30/23 21:54 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

4.5.1: Inserting nodes into a basic linked list.

**Animation captions:**

1. The headObj pointer points to a special node that represents the front of the list. When the list is first created, no list items exist, so the head node's nextNodePtr pointer is null. ©zyBooks 05/30/23 21:54 1692462 Taylor Larrechea COLORADOCSPB2270Summer2023
2. To insert a node in the list, the new node nodeObj1 is first created with the value 555.
3. To insert the new node, tmpNext is pointed to the head node's next node, the head node's nextNodePtr is pointed to the new node, and the new node's nextNodePtr is pointed to tmpNext.
4. A second node nodeObj2 with the value 999 is inserted at the end of the list, and a third node nodeObj3 with the value 777 is created.
5. To insert nodeObj3 after nodeObj1, tmpNext is pointed to the nodeObj1's next node, the nodeObj1's nextNodePtr is pointed to the nodeObj3, and nodeObj3's nextNodePtr is pointed to tmpNext.

The most interesting part of the above program is the InsertAfter() function, which inserts a new node after a given node already in the list. The above animation illustrates.

PARTICIPATION ACTIVITY

4.5.2: A first linked list.



Some questions refer to the above linked list code and animation.

- 1) A linked list has what key advantage over a sequential storage approach like an array or vector?

- An item can be inserted somewhere in the middle of the list without having to shift all subsequent items.
- Uses less memory overall.
- Can store items other than int variables.

- 2) What is the purpose of a list's head node?

- Stores the first item in the list.
- Provides a pointer to the first item's node in the list, if such an item exists.
- Stores all the data of the list.

- 3) After the above list is done having items inserted, at what memory address is the last list item's node located?

- 80
- 82

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

84 86

- 4) After the above list has items inserted as above, if a fourth item was inserted at the front of the list, what would happen to the location of node1?

- Changes from 84 to 86.
- Changes from 84 to 82.
- Stays at 84.



©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

In contrast to the above program that declares one variable for each item allocated by the new operator, a program commonly declares just one or a few variables to manage a large number of items allocated using the new operator. The following example replaces the above main() function, showing how just two pointer variables, currObj and lastObj, can manage 20 allocated items in the list.

To run the following figure, `#include <cstdlib>` was added to access the `rand()` function.

Figure 4.5.2: Managing many new items using just a few pointer variables.

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

```

#include <iostream>
#include <cstdlib>
using namespace std;

class IntNode {
public:
    IntNode(int dataInit = 0, IntNode* nextLoc = nullptr);
    void InsertAfter(IntNode* nodeLoc);
    IntNode* GetNext();
    void PrintNodeData();
private:
    int dataVal;
    IntNode* nextNodePtr;
};

// Constructor
IntNode::IntNode(int dataInit, IntNode* nextLoc) {
    this->dataVal = dataInit;
    this->nextNodePtr = nextLoc;
}

/* Insert node after this node.
 * Before: this -- next
 * After:  this -- node -- next
 */
void IntNode::InsertAfter(IntNode* nodeLoc) {
    IntNode* tmpNext = nullptr;

    tmpNext = this->nextNodePtr; // Remember next
    this->nextNodePtr = nodeLoc; // this -- node -- ?
    nodeLoc->nextNodePtr = tmpNext; // this -- node -- next
}

// Print dataVal
void IntNode::PrintNodeData() {
    cout << this->dataVal << endl;
}

// Grab location pointed by nextNodePtr
IntNode* IntNode::GetNext() {
    return this->nextNodePtr;
}

int main() {
    IntNode* headObj = nullptr; // Create IntNode pointers
    IntNode* currObj = nullptr;
    IntNode* lastObj = nullptr;
    int i; // Loop index

    headObj = new IntNode(-1); // Front of nodes list
    lastObj = headObj;

    for (i = 0; i < 20; ++i) { // Append 20 rand nums
        currObj = new IntNode(rand());

        lastObj->InsertAfter(currObj); // Append curr
        lastObj = currObj; // Curr is the new last item
    }

    currObj = headObj; // Print the list

    while (currObj != nullptr) {
        currObj->PrintNodeData();
        currObj = currObj->GetNext();
    }

    return 0;
}

```

©zyBooks 05/30/23 21:54 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

-1
16807
282475249
1622650073
984943658
1144108930
470211272
101027544
1457850878
1458777923
2007237709
823564440
1115438165
1784484492
74243042
114807987
1137522503
1441282327
16531729
823378840
143542612

©zyBooks 05/30/23 21:54 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

zyDE 4.5.1: Managing a linked list.

Finish the program so that it finds and prints the smallest value in the linked list.

Load default template...

Run

```
1 #include <iostream>
2 #include <cstdlib>
3 using namespace std;
4
5 class IntNode {
6 public:
7     IntNode(int dataInit = 0, IntNod
8     void InsertAfter(IntNode* nodeLo
9     IntNode* GetNext();
10    void PrintNodeData();
11    int GetDataVal();
12 private:
13    int dataVal;
14    IntNode* nextNodePtr;
15};
```

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Normally, a linked list would be maintained by member functions of another class, such as IntList. Private data members of that class might include the list head (a list node allocated by the list class constructor), the list size, and the list tail (the last node in the list). Public member functions might include InsertAfter (insert a new node after the given node), PushBack (insert a new node after the last node), PushFront (insert a new node at the front of the list, just after the head), DeleteNode (deletes the node from the list), etc.

Exploring further:

- [More on Linked Lists](#) from cplusplus.com

CHALLENGE
ACTIVITY

4.5.1: Enter the output of the program using Linked List.



489394.3384924.qx3zqy7

Start

Type the program's output

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

```

#include <iostream>
using namespace std;

class PlaylistSong {
public:
    PlaylistSong(string value = "noName", PlaylistSong* nextLoc = nullptr);
    void InsertAfter(PlaylistSong* nodePtr);
    PlaylistSong* GetNext();
    void PrintNodeData();
private:
    string name;
    PlaylistSong* nextPlaylistSongPtr;
};

PlaylistSong::PlaylistSong(string name, PlaylistSong* nextLoc) {
    this->name = name;
    this->nextPlaylistSongPtr = nextLoc;
}

void PlaylistSong::InsertAfter(PlaylistSong* nodeLoc) {
    PlaylistSong* tmpNext = nullptr;

    tmpNext = this->nextPlaylistSongPtr;
    this->nextPlaylistSongPtr = nodeLoc;
    nodeLoc->nextPlaylistSongPtr = tmpNext;
}

PlaylistSong* PlaylistSong::GetNext() {
    return this->nextPlaylistSongPtr;
}

void PlaylistSong::PrintNodeData() {
    cout << this->name << endl;
}

int main() {
    PlaylistSong* headObj = nullptr;
    PlaylistSong* firstSong = nullptr;
    PlaylistSong* secondSong = nullptr;
    PlaylistSong* thirdSong = nullptr;
    PlaylistSong* currObj = nullptr;

    headObj = new PlaylistSong("head");

    firstSong = new PlaylistSong("Pavanne");
    headObj->InsertAfter(firstSong);

    secondSong = new PlaylistSong("Vocalise");
    firstSong->InsertAfter(secondSong);

    thirdSong = new PlaylistSong("Canon");
    secondSong->InsertAfter(thirdSong);

    currObj = headObj;

    while (currObj != nullptr) {
        currObj->PrintNodeData();
        currObj = currObj->GetNext();
    }
    return 0;
}

```

©zyBooks 05/30/23 21:54 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

1

2

Check**Next**

©zyBooks 05/30/23 21:54 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

CHALLENGE ACTIVITY

4.5.2: A first linked list.

489394.3384924.qx3zqy7

Start

Two integers, neighbors1 and neighbors2, are read from input as the number of neighbors of two towns. The value of -1. Create a new node firstTown with integer neighbors1 and insert firstTown after headObj. Then create a new node secondTown with integer neighbors2 and insert secondTown after firstTown.

Ex: If the input is 24 11, then the output is:

```
-1
24
11
```

```
1 #include <iostream>
2 using namespace std;
3
4 class TownNode {
5     public:
6         TownNode(int neighborsInit = 0, TownNode* nextLoc = nullptr);
7         void InsertAfter(TownNode* nodeLoc);
8         TownNode* GetNext();
9         void PrintNodeData();
10    private:
11        int neighborsVal;
12        TownNode* nextNodePtr;
13    };
14
15 TownNode::TownNode(int neighborsInit, TownNode* nextLoc) {
```

©zyBooks 05/30/23 21:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

1

2

Check

Next level

4.6 Memory regions: Heap/Stack

A program's memory usage typically includes four different regions:

- **Code** – The region where the program instructions are stored.
- **Static memory** – The region where global variables (variables declared outside any function) as well as static local variables (variables declared inside functions starting with the keyword "static") are allocated. Static variables are allocated once and stay in the same memory location for the duration of a program's execution.
- **The stack** – The region where a function's local variables are allocated during a function call. A function call adds local variables to the stack, and a return removes them, like adding and removing dishes from a pile; hence the term "stack." Because this memory is automatically allocated and deallocated, it is also called **automatic memory**.
- **The heap** – The region where the "new" operator allocates memory, and where the "delete" operator deallocates memory. The region is also called **free store**.

PARTICIPATION
ACTIVITY

4.6.1: Use of the four memory regions.

©zyBooks 05/30/23 21:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Animation captions:

1. The code regions store program instructions. myGlobal is a global variable and is stored in the static memory region. Code and static regions last for the entire program execution.
2. Function calls push local variables on the program stack. When main() is called, the variables myInt and myPtr are added on the stack.
3. new allocates memory on the heap for an int and returns the address of the allocated memory, which is assigned to myPtr. delete deallocates memory from the heap.

4. Calling MyFct() grows the stack, pushing the function's local variables on the stack. Those local variables are removed from the stack when the function returns.
5. When main() completes, main's local variables are removed from the stack.

PARTICIPATION ACTIVITY**4.6.2: Stack and heap definitions.**

If unable to drag and drop, refresh the page.

©zyBooks 05/30/23 21:54 1692462

Taylor Larrechea

SPB2270Summer2023

Static memory

The stack

Automatic memory

The heap

Code

Free store

A function's local variables are allocated in this region while a function is called.

The memory allocation and deallocation operators affect this region.

Global and static local variables are allocated in this region once for the duration of the program.

Another name for "The heap" because the programmer has explicit control of this memory.

Instructions are stored in this region.

Another name for "The stack" because the programmer does not explicitly control this memory.

Reset

4.7 Memory leaks

Memory leak

A **memory leak** occurs when a program that allocates memory loses the ability to access the allocated memory, typically due to failure to properly destroy/free dynamically allocated memory. A program's leaking memory becomes unusable, much like a water pipe might have water leaking out and becoming unusable. A memory leak may cause a program to occupy more and more memory as the program runs, which slows program runtime. Even worse, a memory leak can cause the program to fail if memory becomes completely full and the program is unable to allocate additional memory.

A common error is failing to free allocated memory that is no longer used, resulting in a memory leak. Many programs that are commonly left running for long periods, like web browsers, suffer from known memory leaks – a web search for "<your-favorite-browser> memory leak" will likely result in numerous hits.

PARTICIPATION ACTIVITY

4.7.1: Memory leak can use up all available memory.

**Animation captions:**

1. Memory is allocated for newVal each loop iteration, but the loop does not deallocate memory once done using newVal, resulting in a memory leak.
2. Each loop iteration allocates more memory, eventually using up all available memory and causing the program to fail.

©zyBooks 05/30/23 21:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Garbage collection

Some programming languages, such as Java, use a mechanism called **garbage collection** wherein a program's executable includes automatic behavior that at various intervals finds all unreachable allocated memory locations (e.g., by comparing all reachable memory with all previously-allocated memory), and automatically frees such unreachable memory. Some non-standard C++ implementations also include garbage collection. Garbage collection can reduce the impact of memory leaks at the expense of runtime overhead. Computer scientists debate whether new programmers should learn to explicitly free memory versus letting garbage collection do the work.

PARTICIPATION ACTIVITY

4.7.2: Memory leaks.



If unable to drag and drop, refresh the page.

Unusable memory**Memory leak****Garbage collection**

Memory locations that have been dynamically allocated but can no longer be used by a program.

Occurs when a program allocates memory but loses the ability to access the allocated memory.

Automatic process of finding and freeing unreachable allocated memory locations.

©zyBooks 05/30/23 21:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Reset**Memory not freed in a destructor**

Destructors are needed when destroying an object involves more work than simply freeing the object's memory. Such a need commonly arises when an object's data member, referred to as a sub-object, has allocated additional memory. Freeing the

object's memory without also freeing the sub-object's memory results in a problem where the sub-object's memory is still allocated, but inaccessible, and thus can't be used again by the program.

The program in the animation below is very simple to focus on how memory leaks occur with sub-objects. The class's sub-object is just an integer pointer but typically would be a pointer to a more complex type. Likewise, the object is created and then immediately destroyed, but typically something would have been done with the object.

PARTICIPATION ACTIVITY

4.7.3: Lack of destructor yields memory leak.



©zyBooks 05/30/23 21:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Animation content:

undefined

Animation captions:

1. tempClassObject is a pointer to object of type MyClass. new allocates memory for the object.
2. The constructor for the MyClass object is called. The constructor allocates memory for an int using the pointer subObject.
3. Deleting tempClassObject frees the memory for the tempClassObject, but not subObject. A memory leak results because memory location 78 is still allocated, but nothing points to the memory allocation.

PARTICIPATION ACTIVITY

4.7.4: Memory not freed in a destructor.



- 1) In the above animation, which object's memory is not freed?

- MyClass
- tempClassObject
- subObject

- 2) Does a memory leak remain when the above program terminates?

- Yes
- No

- 3) What line must exist in MyClass's destructor to free all memory allocated by a MyClass object?

- delete subObject;
- delete tempClassObject;
- delete MyClass;

©zyBooks 05/30/23 21:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

4.7.5: Which results in a memory leak?



Which scenario results in a memory leak?



```
1) int main() {
    MyClass* ptrOne = new
MyClass;
    MyClass* ptrTwo = new
MyClass;

    ptrOne = ptrTwo;
    return 0;
}
```

- Memory leak
- No memory leak

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



```
2) int main() {
    MyClass* ptrOne = new
MyClass;
    MyClass* ptrTwo = new
MyClass;
    MyClass* ptrThree;

    ptrThree = ptrOne;
    ptrOne = ptrTwo;
    return 0;
}
```

- Memory leak
- No memory leak



```
3) class MyClass {
public:
    MyClass() {
        subObject = new int;
        *subObject = 0;
    }

    ~MyClass() {
        delete subObject;
    }

private:
    int* subObject;
};

int main() {
    MyClass* ptrOne = new
MyClass;
    MyClass* ptrTwo = new
MyClass;
    ...

    delete ptrOne;
    ptrOne = ptrTwo;
    return 0;
}
```

- Memory leak
- No memory leak

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

4.8 Destructors

Overview

A **destructor** is a special class member function that is called automatically when a variable of that class type is destroyed. C++ class objects commonly use dynamically allocated data that is deallocated by the class's destructor.

Ex: A linked list class dynamically allocates nodes when adding items to the list. Without a destructor, the link list's nodes are not deallocated. The linked list class destructor should be implemented to deallocate each node in the list.

PARTICIPATION ACTIVITY

4.8.1: LinkedList nodes are not deallocated without a LinkedList class destructor.



©zyBooks 05/30/23 21:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Animation content:

undefined

Animation captions:

1. The LinkedList class has a pointer to the list's head, initially set to null by the LinkedList class constructor.
2. After adding 2 items, 3 dynamically allocated objects exist: the list itself and 2 nodes.
3. Without a destructor, deleting list1 only deallocates the list1 object, but not the 2 nodes.
4. With a properly implemented destructor, the LinkedList class will free the list's nodes.

PARTICIPATION ACTIVITY

4.8.2: LinkedList class destructor.



- 1) Using the delete operator to deallocate a LinkedList object automatically frees all nodes allocated by that object.

- True
- False

- 2) A destructor for the LinkedList class would be implemented as a LinkedList class member function.

- True
- False

- 3) If list1 were declared without dynamic allocation, as shown below, no destructor would be needed.

```
LinkedList list1;
```



- True
- False



©zyBooks 05/30/23 21:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Implementing the LinkedList class destructor

The syntax for a class's destructor function is similar to a class's constructor function, but with a "~" (called a "tilde" character) prepended to the function name. A destructor has no parameters and no return value. So the `LinkedListNode` and `LinkedList` class destructors are declared as `~LinkedListNode();` and `~LinkedList();`, respectively.

The `LinkedList` class destructor is implemented to free each node in the list. The `LinkedListNode` destructor is not required, but is implemented below to display a message when a node's destructor is called. Using `delete` to free a dynamically allocated `LinkedListNode` or `LinkedList` will call the object's destructor.

Figure 4.8.1: LinkedListNode and LinkedList classes.

```
#include <iostream>
using namespace std;

class LinkedListNode {
public:
    LinkedListNode(int dataValue) {
        cout << "In LinkedListNode constructor (" << dataValue << ")" << endl;
        data = dataValue;
    }

    ~LinkedListNode() {
        cout << "In LinkedListNode destructor (" << data << ")" << endl;
    }

    int data;
    LinkedListNode* next;
};

class LinkedList {
public:
    LinkedList();
    ~LinkedList();
    void Prepend(int dataValue);

    LinkedListNode* head;
};

LinkedList::LinkedList() {
    cout << "In LinkedList constructor" << endl;
    head = nullptr;
}

LinkedList::~LinkedList() {
    cout << "In LinkedList destructor" << endl;

    // The destructor deletes each node in the linked list
    while (head) {
        LinkedListNode* next = head->next;
        delete head;
        head = next;
    }
}

void LinkedList::Prepend(int dataValue) {
    LinkedListNode* newNode = new LinkedListNode(dataValue);
    newNode->next = head;
    head = newNode;
}
```

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION
ACTIVITY

4.8.3: The LinkedList class destructor, called when the list is deleted, frees all nodes.

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Animation content:

undefined

Animation captions:

1. A linked list is created and 5 dynamically allocated nodes are prepended.
2. Deleting the list calls the `LinkedList` class destructor.
3. The destructor deletes each node in the list.
4. After calling `~LinkedList()`, the delete operator frees memory for the linked list's head pointer. All memory for the linked list has been freed.

PARTICIPATION ACTIVITY**4.8.4: `LinkedList` class destructor.**

@zyBooks 05/30/23 21:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

- 1) After `~LinkedList()` is called, the list's head pointer points to ____.
 - null
 - the first node, which is now freed
 - the last node, which is now freed
- 2) When `~LinkedList()` is called, `~LinkedListNode()` gets called for each node in the list.
 - True
 - False
- 3) If the `LinkedList` class were renamed to just `List`, the destructor function must be redeclared as ____.
 - `void ~List();`
 - `~List();`
 - `List();`

**When a destructor is called**

Using the delete operator on an object allocated with the new operator calls the destructor, as shown in the previous example. For an object that is not declared by reference or by pointer, the object's destructor is called automatically when the object goes out of scope.

PARTICIPATION ACTIVITY**4.8.5: Destructors are called automatically only for non-reference/pointer variables.****Animation content:**

undefined

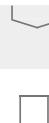
@zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023**Animation captions:**

1. `list1` is declared as a local variable and is not a pointer or reference.
2. `list1` goes out of scope at the end of `main()`. So `list1`'s destructor is called automatically.
3. `list2` is declared as a pointer and the destructor is not automatically called at the end of `main()`.
4. `list3` is declared as a reference and the destructor is not automatically called at the end of `main()`.



**PARTICIPATION
ACTIVITY**

4.8.6: When a destructor is called.



- 1) Both the constructor and destructor are called by the following code.

```
delete (new LinkedList());
```

- True
 False

- 2) listToDisplay's destructor is called at the end of the DisplayList function.

```
void DisplayList(LinkedList  
listToDisplay) {  
    LinkedListNode* node =  
listToDisplay.head;  
    while(node) {  
        cout << node->data << " ";  
        node = node->next;  
    }  
}
```

- True
 False

- 3) listToDisplay's destructor is called at the end of the DisplayList function.

```
void DisplayList(LinkedList&  
listToDisplay) {  
    LinkedListNode* node =  
listToDisplay.head;  
    while(node) {  
        cout << node->data << " ";  
        node = node->next;  
    }  
}
```

- True
 False

**CHALLENGE
ACTIVITY**

4.8.1: Enter the output of the destructors.



489394.3384924.qx3zqy7

Start

Type the program's output

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

```
#include <iostream>
using namespace std;

class IntNode {
public:
    IntNode(int value) {
        numVal = value;
    }

    ~IntNode() {
        cout << numVal << endl;
    }

    int numVal;
};

int main() {
    IntNode* node1 = new IntNode(1);
    IntNode* node2 = new IntNode(4);
    IntNode* node3 = new IntNode(6);
    IntNode* node4 = new IntNode(7);

    delete node2;
    delete node1;
    delete node3;
    delete node4;

    return 0;
}
```



©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

1

2

Check**Next**
CHALLENGE ACTIVITY
4.8.2: Destructors.


489394.3384924.qx3zqy7

Start

Complete the TownNode class destructor. The destructor prints "Deallocating TownNode (" followed by a space, and the value of population, and then ")". End with a newline.

Ex: If the input is Opal 4106, then the output is:

Deallocating TownNode (Opal 4106)

```
1 #include <iostream>
2 using namespace std;
3
4 class TownNode {
5 public:
6     TownNode(string nameValue, int populationValue) {
7         name = nameValue;
8         population = populationValue;
9     }
10
11     /* Your code goes here */
12
13     string name;
14     int population;
15     TownNode* next;
16 }
```

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

1

2

[Check](#)[Next level](#)

Exploring further:

- [More on Destructors](#) from msdn.microsoft.com
- [Order of Destruction](#) from msdn.microsoft.com

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

4.9 Copy constructors

Copying an object without a copy constructor

The animation below shows a typical problem that arises when an object is passed by value to a function and no copy constructor exists for the object.

PARTICIPATION
ACTIVITY

4.9.1: Copying an object without a copy constructor.



Animation content:

undefined

Animation captions:

1. The constructor creates object tempClassObject and sets the object's dataObject (a pointer) to the value 9. The value 9 is printed.
2. SomeFunction() is called and tempClassObject is passed by value, creating a local copy of the object with the same dataObject.
3. When SomeFunction() returns, localObject is destroyed and the MyClass destructor frees the dataObject's memory. tempClassObject's dataObject value is changed and now 0 is printed.
4. When main() returns, the MyClass destructor is called again, attempting to free the dataObject's memory again, causing the program to crash.

PARTICIPATION
ACTIVITY

4.9.2: Copying an object without a copy constructor.



- 1) If an object with an int sub-object is passed by value to a function, the program will complete execution with no errors.

- True
 False

- 2) If an object with an int* sub-object is passed by value to a function, the program will complete execution with no errors.

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- True
- False

3) If an object with an int* sub-object is passed by value to a function, the program will call the class constructor to create a local copy of the sub-object.

- True
- False

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Copy constructor

The solution is to create a **copy constructor**, a constructor that is automatically called when an object of the class type is passed by value to a function and when an object is initialized by copying another object during declaration. Ex:

`MyClass classObj2 = classObj1;` or `obj2Ptr = new MyClass(classObj1);`. The copy constructor makes a new copy of all data members (including pointers), known as a **deep copy**.

If the programmer doesn't define a copy constructor, then the compiler implicitly defines a constructor with statements that perform a memberwise copy, which simply copies each member using assignment:

`newObj.member1 = origObj.member1, newObj.member2 = origObj.member2`, etc. Creating a copy of an object by copying only the data members' values creates a **shallow copy** of the object. A shallow copy is fine for many classes, but typically a deep copy is desired for objects that have data members pointing to dynamically allocated memory.

The copy constructor can be called with a single pass-by-reference argument of the class type, representing an original object to be copied to the newly-created object. A programmer may define a copy constructor, typically having the form:

`MyClass(const MyClass& origObject);`

Construct 4.9.1: Copy constructor.

```
class MyClass {
public:
    ...
    MyClass(const MyClass&
origObject);
    ...
};
```

The program below adds a copy constructor to the earlier example, which makes a deep copy of the data member `dataObject` within the `MyClass` object. The copy constructor is automatically called during the call to `SomeFunction()`. Destruction of the local object upon return from `SomeFunction()` frees the newly created `dataObject` for the local object, leaving the original `tempClassObject`'s `dataObject` untouched. Printing after the function call correctly prints 9, and destruction of `tempClassObject` during the return from `main()` produces no error.

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Figure 4.9.1: Problem solved by creating a copy constructor that does a deep copy.

```
#include <iostream>
using namespace std;

class MyClass {
public:
    MyClass();
    MyClass(const MyClass& origObject); // Copy constructor
    ~MyClass();

    // Set member value dataObject
    void SetDataObject(const int setVal) {
        *dataObject = setVal;
    }

    // Return member value dataObject
    int GetDataObject() const {
        return *dataObject;
    }
private:
    int* dataObject; // Data member
};

// Default constructor
MyClass::MyClass() {
    cout << "Constructor called." << endl;
    dataObject = new int; // Allocate mem for data
    *dataObject = 0;
}

// Copy constructor
MyClass::MyClass(const MyClass& origObject) {
    cout << "Copy constructor called." << endl;
    dataObject = new int; // Allocate sub-object
    *dataObject = *(origObject.dataObject);
}

// Destructor
MyClass::~MyClass() {
    cout << "Destructor called." << endl;
    delete dataObject;
}

void SomeFunction(MyClass localObj) {
    // Do something with localObj
}

int main() {
    MyClass tempClassObject; // Create object of type MyClass

    // Set and print data member value
    tempClassObject.SetDataObject(9);
    cout << "Before: " << tempClassObject.GetDataObject() << endl;

    // Calls SomeFunction(), tempClassObject is passed by value
    SomeFunction(tempClassObject);

    // Print data member value
    cout << "After: " << tempClassObject.GetDataObject() << endl;

    return 0;
}
```

Constructor called.
Before: 9
Copy constructor called.
Destructor called.
After: 9
Destructor called.

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Copy constructors in more complicated situations

The above examples use a trivially-simple class having a `dataObject` whose type is a pointer to an integer, to focus attention on the key issue. Real situations typically involve classes with multiple data members and with data objects whose types are pointers to class-type objects.

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

4.9.3: Determining which constructor will be called.



Given the following class declaration and variable declaration, determine which constructor will be called for each of the following statements.

```
class EncBlock {
public:
    EncBlock();                                // Default constructor
    EncBlock(const EncBlock& origObj);        // Copy constructor
    EncBlock(int blockSize);                   // Constructor with int parameter
    ~EncBlock();                               // Destructor
    ...
};

EncBlock myBlock;
```

1) `EncBlock* aBlock = new EncBlock(5);`

- `EncBlock();`
- `EncBlock(const EncBlock& origObj);`
- `EncBlock(int blockSize);`



2) `EncBlock testBlock;`

- `EncBlock();`
- `EncBlock(const EncBlock& origObj);`
- `EncBlock(int blockSize);`



3) `EncBlock* lastBlock = new EncBlock(myBlock);`

- `EncBlock();`
- `EncBlock(const EncBlock& origObj);`
- `EncBlock(int blockSize);`



4) `EncBlock vidBlock = myBlock;`

- `EncBlock();`
- `EncBlock(const EncBlock& origObj);`
- `EncBlock(int blockSize);`

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Exploring further:

- [More on Copy Constructors](#) from cplusplus.com

CHALLENGE ACTIVITY

4.9.1: Enter the output of the copy constructors.



@zyBooks 05/30/23 21:54 1692462

Taylor Larrechea

COLORADO CSPB2270Summer2023

489394.3384924.qx3zqy7

Start

Type the program's output

```
#include <iostream>
using namespace std;

class IntNode {
public:
    IntNode(int value) {
        numVal = new int;
        *numVal = value;
    }
    void SetNumVal(int val) { *numVal = val; }
    int GetNumVal() { return *numVal; }
private:
    int* numVal;
};

int main() {
    IntNode node1(1);
    IntNode node2(2);
    IntNode node3(3);

    node3 = node2;
    node2.SetNumVal(9);

    cout << node3.GetNumVal() << " " << node2.GetNumVal() << endl;

    return 0;
}
```



1

2

Check

Next

CHALLENGE ACTIVITY

4.9.2: Write a copy constructor.



Write a copy constructor for CarCounter that assigns origCarCounter.carCount to the constructed object's carCount. Sample output for the given program:

Cars counted: 5

@zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADO CSPB2270Summer2023

[Learn how our autograder works](#)

489394.3384924.qx3zqy7

```
1 #include <iostream>
2 using namespace std;
3
4 class CarCounter {
```

```

5   public:
6     CarCounter();
7     CarCounter(const CarCounter& origCarCounter);
8     void SetCarCount(const int count) {
9       carCount = count;
10    }
11    int GetCarCount() const {
12      return carCount;
13    }
14  private:
15    int carCount;

```

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Run

View your last submission ▾

CHALLENGE
ACTIVITY

4.9.3: Copy constructors.



489394.3384924.qx3zqy7

Start

SavingsAccount is a class with two double* data members pointing to the amount saved and interest respectively. Two doubles are read from input to initialize userAccount. Use the copy constructor to create an object named copyAccount that is a deep copy of userAccount.

Ex: If the input is 80.00 0.03, then the output is:

```

Original constructor called
Called SavingsAccount's copy constructor
userAccount: $80.00 with 3.00% interest rate
copyAccount: $160.00 with 6.00% interest rate

```

```

1 #include <iostream>
2 #include <iomanip>
3 using namespace std;
4
5 class SavingsAccount {
6 public:
7     SavingsAccount(double startingSaved = 0.0, double startingRate = 0.0);
8     SavingsAccount(const SavingsAccount& acc);
9     void SetSaved(double newSaved);
10    void SetRate(double newRate);
11    double GetSaved() const;
12    double GetRate() const;
13    void Print() const;
14  private:
15    double* saved;

```

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

1

2

Check**Next level**

4.10 Copy assignment operator

Default assignment operator behavior

Given two MyClass objects, classObj1 and classObj2, a programmer might write `classObj2 = classObj1;` to copy classObj1 to classObj2. The default behavior of the assignment operator (=) for classes or structs is to perform memberwise assignment. Ex:

```
classObj2.memberVal1 = classObj1.memberVal1;
classObj2.memberVal2 = classObj1.memberVal2;
...
```

©zyBooks 05/30/23 21:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Such behavior may work fine for members with basic types like int or char, but typically is not the desired behavior for a pointer member. Memberwise assignment of pointers may lead to program crashes or memory leaks.

PARTICIPATION ACTIVITY

4.10.1: Basic assignment operation fails when pointer member involved.



Animation content:

undefined

Animation captions:

1. Two MyClass objects, classObject1 and classObject2, are created. classObject1's SetDataObject() function assigns the memory location pointed to by dataObject with 9.
2. The assignment classObject2 = classObject1; copies the pointer for classObject1's dataObject to classObject2, resulting in both dataObject members pointing to the same memory location.
3. Destroying classObject1 frees that object's memory.
4. Destroying classObject2 then tries to free that same memory, causing a program crash. A memory leak also occurs because neither object is pointing to location 81.

PARTICIPATION ACTIVITY

4.10.2: Default assignment operator behavior.



- 1) The default assignment operator often works for objects without pointer members.

- True
- False



- 2) When used with objects with pointer members, the default assignment operator behavior may lead to crashes due to the same memory being freed more than once.

- True
- False



©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 3) When used with objects with pointer members, the default assignment operator behavior may lead to memory leaks.



- True
- False

Overloading the assignment operator

The assignment operator (=) can be overloaded to eliminate problems caused by a memberwise assignment during an object copy. The implementation of the assignment operator iterates through each member of the source object. Each non-pointer member is copied directly from source member to destination member. For each pointer member, new memory is allocated, the source's referenced data is copied to the new memory, and a pointer to the new member is assigned as the destination member. Allocating and copying data for pointer members is known as a **deep copy**.

The following program solves the default assignment operator behavior problem by introducing an assignment operator that performs a deep copy.

Figure 4.10.1: Assignment operator performs a deep copy.

```
#include <iostream>
using namespace std;

class MyClass {
public:
    MyClass();
    ~MyClass();
    MyClass& operator=(const MyClass& objToCopy);

    // Set member value dataObject
    void SetDataObject(const int setVal) {
        *dataObject = setVal;
    }

    // Return member value dataObject
    int GetDataObject() const {
        return *dataObject;
    }
private:
    int* dataObject;// Data member
};

// Default constructor
MyClass::MyClass() {
    cout << "Constructor called." << endl;
    dataObject = new int; // Allocate mem for data
    *dataObject = 0;
}

// Destructor
MyClass::~MyClass() {
    cout << "Destructor called." << endl;
    delete dataObject;
}

MyClass& MyClass::operator=(const MyClass& objToCopy) {
    cout << "Assignment op called." << endl;

    if (this != &objToCopy) {           // 1. Don't self-assign
        delete dataObject;           // 2. Delete old dataObject
        dataObject = new int;         // 3. Allocate new dataObject
        *dataObject = *(objToCopy.dataObject); // 4. Copy dataObject
    }

    return *this;
}

int main() {
    MyClass classObj1; // Create object of type MyClass
    MyClass classObj2; // Create object of type MyClass

    // Set and print object 1 data member value
    classObj1.SetDataObject(9);

    // Copy class object using copy assignment operator
    classObj2 = classObj1;

    // Set object 1 data member value
    classObj1.SetDataObject(1);

    // Print data values for each object
    cout << "classObj1:" << classObj1.GetDataObject() << endl;
    cout << "classObj2:" << classObj2.GetDataObject() << endl;

    return 0;
}
```

©zyBooks 05/30/23 21:54 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

©zyBooks 05/30/23 21:54 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

```
Constructor called.  
Constructor called.  
Assignment op called.  
obj1:1  
obj2:9  
Destructor called.  
Destructor called.
```

PARTICIPATION ACTIVITY

4.10.3: Assignment operator.

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



- 1) Declare a copy assignment operator for a class named `EngineMap` using `inVal` as the input parameter name.

```
EngineMap& operator=(  
    _____);
```

Check**Show answer**

- 2) Provide the return statement for the copy assignment operator for the `EngineMap` class.

```
    _____;
```

Check**Show answer****CHALLENGE ACTIVITY**

4.10.1: Enter the output of the program with an overloaded assignment operator.



489394.3384924.qx3zqy7

Start

Type the program's output

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

```
#include <iostream>
using namespace std;

class SubstituteTeacher {
public:
    SubstituteTeacher();
    ~SubstituteTeacher();
    SubstituteTeacher& operator=(const SubstituteTeacher& objToCopy);

    void SetSubject(const string& setVal) {
        *subject = setVal;
    }

    string GetSubject() const {
        return *subject;
    }
private:
    string* subject;
};

SubstituteTeacher::SubstituteTeacher() {
    subject = new string;
    *subject = "none";
}

SubstituteTeacher::~SubstituteTeacher() {
    delete subject;
}

SubstituteTeacher& SubstituteTeacher::operator=(const SubstituteTeacher& objToCopy) {
    if (this != &objToCopy) {
        delete subject;
        subject = new string;
        *subject = *(objToCopy.subject);
    }

    return *this;
}

int main() {
    SubstituteTeacher msDorf;
    SubstituteTeacher mrDiaz;
    SubstituteTeacher msPark;

    msPark.SetSubject("English");
    mrDiaz = msPark;
    mrDiaz.SetSubject("Art");
    msDorf = mrDiaz;

    cout << msPark.GetSubject() << endl;
    cout << msDorf.GetSubject() << endl;

    return 0;
}
```

©zyBooks 05/30/23 21:54 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

1

2

Check**Next**
CHALLENGE
ACTIVITY

4.10.2: Copy assignment operator.



489394.3384924.qx3zqy7

©zyBooks 05/30/23 21:54 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

Complete the copy assignment operator to prevent self-assignment.

Ex: If the input is 3.30, then the output is:

```
Self-assignment not permitted
account1: 1.70% rate
copyAccount1: 3.30% rate
```

Destructor called
Destructor called

```

1 #include <iostream>
2 #include <iomanip>
3 using namespace std;
4
5 class SavingsAccount {
6 public:
7     SavingsAccount();
8     ~SavingsAccount();
9     void setRatePercent(double newRatePercent);
10    void Print() const;
11    SavingsAccount& operator=(const SavingsAccount& accountToCopy);
12 private:
13     double* ratePercent;
14 };
15

```

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

1

2

Check**Next level**

4.11 Rule of three

Classes have three special member functions that are commonly implemented together:

- **Destructor:** A destructor is a class member function that is automatically called when an object of the class is destroyed, such as when the object goes out of scope or is explicitly destroyed as in `delete someObject;`.
- **Copy constructor:** A copy constructor is another version of a constructor that can be called with a single pass by reference argument. The copy constructor is automatically called when an object is passed by value to a function, such as for the function `SomeFunction(MyClass localObject)` and the call `SomeFunction(anotherObject)`, when an object is initialized when declared such as `MyClass localObject1 = localObject2;`, or when an object is initialized when allocated via "new" as in `newObjectPtr = new MyClass(classObject2);`.
- **Copy assignment operator:** The assignment operator "=" can be overloaded for a class via a member function, known as the copy assignment operator, that overloads the built-in function "operator=", the member function having a reference parameter of the class type and returning a reference to the class type.

The **rule of three** describes a practice that if a programmer explicitly defines any one of those three special member functions (destructor, copy constructor, copy assignment operator), then the programmer should explicitly define all three. For this reason, those three special member functions are sometimes called **the big three**.

A good practice is to always follow the rule of three and define the big three if any one of these functions are defined.

PARTICIPATION ACTIVITY

4.11.1: Rule of three.

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Animation content:

undefined

Animation captions:

1. The big three consists of the destructor, copy constructor, and copy assignment operator.
2. The default constructor is not part of the big three.
3. A constructor may exist that copies some data, but isn't the copy constructor. The copy constructor for MyClass takes a const MyClass& argument.

Default destructors, copy constructors, and assignment operators

©zyBooks 05/30/23 21:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

- If the programmer doesn't define a destructor for a class, the compiler implicitly defines one having no statements.
- If the programmer doesn't define a copy constructor for a class, then the compiler implicitly defines one whose statements do a memberwise copy, i.e.,
`classObject2.memberVal1 = classObject1.memberVal1,`
`classObject2.memberVal2 = classObject1.memberVal2`, etc.
- If the programmer doesn't define a copy assignment operator, the compiler implicitly defines one that does a memberwise copy.

PARTICIPATION ACTIVITY

4.11.2: Rule of three.



1) If the programmer does not explicitly define a copy constructor for a class, copying objects of that class will not be possible.

- True
 False

2) The big three member functions for classes include a destructor, copy constructor, and default constructor.

- True
 False

3) If a programmer explicitly defines a destructor, copy constructor, or copy assignment operator, it is a good practice to define all three.

- True
 False

4) Assuming `MyClass prevObject` has already been declared, the statement
`MyClass object2 = prevObject;`
will call the copy assignment operator.

- True
 False

5) Assuming `MyClass prevObject` has already been declared, the following

variable declaration will call the copy assignment operator.

```
 MyClass object2;
...
object2 = prevObject;
```

- True
- False

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Exploring further:

- More on [Rule of Three in C++](#) from GeeksforGeeks.

4.12 C++ example: Employee list using vectors

zyDE 4.12.1: Managing an employee list using a vector.

The following program allows a user to add to and list entries from a vector, which maintains a list of employees.

1. Run the program, and provide input to add three employees' names and related data. Then use the list option to display the list.
2. Modify the program to implement the deleteEntry function.
3. Run the program again and add, list, delete, and list again various entries.

[Load default template](#)

```

1 #include <iostream>
2 #include <string>
3 #include <vector>
4 using namespace std;
5
6 // Add an employee
7 void AddEmployee(vector<string> &name, vector<string> &department,
8                  vector<string> &title) {
9     string theName;
10    string theDept;
11    string theTitle;
12
13    cout << endl << "Enter the name to add: " << endl;
14    getline(cin, theName);
15    cout << "Enter " << theName << "'s department: " << endl;
16    cin >> theDept;
17    cout << endl << "Enter " << theName << "'s title: " << endl;
18    cin >> theTitle;
19
20    name.push_back(theName);
21    department.push_back(theDept);
22    title.push_back(theTitle);
23
24    cout << endl << "Employee added successfully!" << endl;
25}

```

a
Rajeev Gupta
Sales
...

Run

Taylor Larrechea
COLORADOCSPB2270Summer2023

Below is a solution to the above problem.

zyDE 4.12.2: Managing an employee list using a vector (solution).

[Load default template](#)

```
1 #include <iostream>
2 #include <string>
3 #include <vector>
4 using namespace std;
5
6
7 // Add an employee
8 void AddEmployee(vector<string> &name, vector<string> &department,
9                  vector<string> &title) {
10    string theName;
11    string theDept;
12    string theTitle;
13
14    cout << endl << "Enter the name to add: " << endl;
15    getline(cin, theName);
16
17    cout << endl << "Enter the department: " << endl;
18    getline(cin, theDept);
19
20    cout << endl << "Enter the title: " << endl;
21    getline(cin, theTitle);
22}
```

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

```
a
Rajeev Gupta
Sales
```

Run

©zyBooks 05/30/23 21:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

5.1 List abstract data type (ADT)

List abstract data type

A **list** is a common ADT for holding ordered data, having operations like append a data item, remove a data item, search whether a data item exists, and print the list. Ex: For a given list item, after "Append 7", "Append 9", and "Append 5", then "Print" will print (7, 9, 5) in that order, and "Search 8" would indicate item not found. A user need not have knowledge of the internal implementation of the list ADT. Examples in this section assume the data items are integers, but a list commonly holds other kinds of data like strings or entire objects.



PARTICIPATION ACTIVITY

5.1.1: List ADT.



Animation captions:

1. A new list named "ages" is created. Items can be appended to the list. The items are ordered.
2. Printing the list prints the items in order.
3. Removing an item keeps the remaining items in order.

PARTICIPATION ACTIVITY

5.1.2: List ADT.



Type the list after the given operations. Each question starts with an empty list. Type the list as: 5, 7, 9

- 1) Append(list, 3)
Append(list, 2)

Check

[Show answer](#)



- 2) Append(list, 3)
Append(list, 2)
Append(list, 1)
Remove(list, 3)

Check

[Show answer](#)



- 3) After the following operations, will
Search(list, 2) find an item? Type yes
or no.

Append(list, 3)

Append(list, 2)

Append(list, 1)

Remove(list, 2)

Check

Show answer

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Common list ADT operations

Table 5.1.1: Some common operations for a list ADT.

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Operation	Description	Example starting with list: 99, 77
Append(list, x)	Inserts x at end of list	Append(list, 44), list: 99, 77, 44
Prepend(list, x)	Inserts x at start of list	Prepend(list, 44), list: 44, 99, 77
InsertAfter(list, w, x)	Inserts x after w	InsertAfter(list, 99, 44), list: 99, 44, ©zyBooks 05/30/23 21:55 1692462 77 Taylor Larrechea COLORADOCSPB2270Summer2023
Remove(list, x)	Removes x	Remove(list, 77), list: 99
Search(list, x)	Returns item if found, else returns null	Search(list, 99), returns item 99 Search(list, 22), returns null
Print(list)	Prints list's items in order	Print(list) outputs: 99, 77
PrintReverse(list)	Prints list's items in reverse order	PrintReverse(list) outputs: 77, 99
Sort(list)	Sorts the lists items in ascending order	list becomes: 77, 99
IsEmpty(list)	Returns true if list has no items	For list 99, 77, IsEmpty(list) returns false
GetLength(list)	Returns the number of items in the list	GetLength(list) returns 2

PARTICIPATION ACTIVITY

5.1.3: List ADT common operations.



- 1) Given a list with items 40, 888, -3, 2, what does GetLength(list) return?

 4 Fails

- 2) Given a list with items 'Z', 'A', 'B', Sort(list) yields 'A', 'B', 'Z'.

 True False

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



- 3) If a list ADT has operations like Sort or PrintReverse, the list is clearly implemented using an array.



- True
- False

5.2 Singly-linked lists

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Singly-linked list data structure

A **singly-linked list** is a data structure for implementing a list ADT, where each node has data and a pointer to the next node. The list structure typically has pointers to the list's first node and last node. A singly-linked list's first node is called the **head**, and the last node the **tail**. A singly-linked list is a type of **positional list**: A list where elements contain pointers to the next and/or previous elements in the list.

null

null is a special value indicating a pointer points to nothing.

The name used to represent a pointer (or reference) that points to nothing varies between programming languages and includes `nil`, `nullptr`, `None`, `NUL`, and even the value `0`.

PARTICIPATION ACTIVITY

5.2.1: Singly-linked list: Each node points to the next node.



Animation content:

undefined

Animation captions:

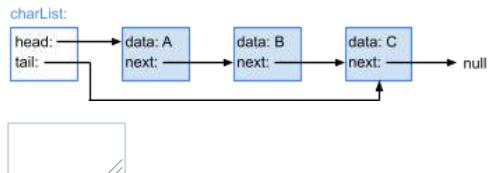
1. A new list item is created, with the head and tail pointers pointing to nothing (null), meaning the list is empty.
2. ListAppend points the list's head and tail pointers to a new node, whose `next` pointer points to null.
3. Another append points the last node's `next` pointer and the list's tail pointer to the new node.
4. Operations like ListAppend, ListPrepend, ListInsertAfter, and ListRemove, update just a few relevant pointers.
5. The list's first node is called the head. The last node is the tail.

**PARTICIPATION
ACTIVITY**

5.2.2: Singly-linked list data structure.



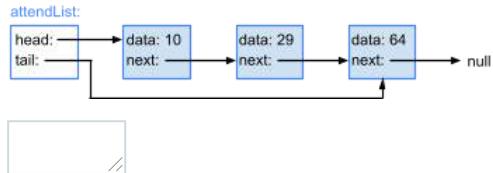
- 1) Given charList, C's next pointer value
is ____.



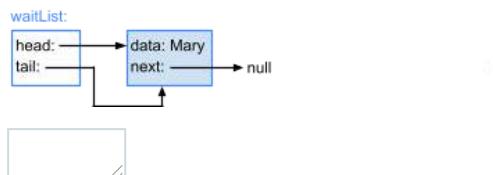
©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Check**Show answer**

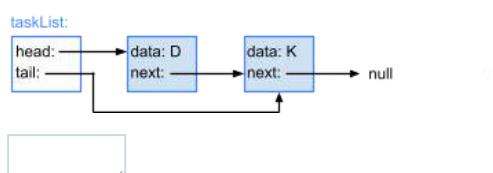
- 2) Given attendList, the head node's data value is ____.
(Answer "None" if no head exists)

**Check****Show answer**

- 3) Given waitList, the tail node's data value is ____.
(Answer "None" if no tail exists)

**Check****Show answer**

- 4) Given taskList, node D is followed by node ____.



©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Check**Show answer**

Appending a node to a singly-linked list

Given a new node, the **Append** operation for a singly-linked list inserts the new node after the list's tail node. Ex: ListAppend(numsList, node 45) appends node 45 to numsList. The notation "node 45" represents a pointer to a node with a data value of 45. This material does not discuss language-specific topics on object creation or memory allocation.

©zyBooks 05/30/23 21:55 1692462

The append algorithm behavior differs if the list is empty versus not empty:

Taylor Larrechea

COLORADO CSPB2270Summer2023

- *Append to empty list*: If the list's head pointer is null (empty), the algorithm points the list's head and tail pointers to the new node.
- *Append to non-empty list*: If the list's head pointer is not null (not empty), the algorithm points the tail node's next pointer and the list's tail pointer to the new node.

PARTICIPATION ACTIVITY

5.2.3: Singly-linked list: Appending a node.



Animation content:

undefined

Animation captions:

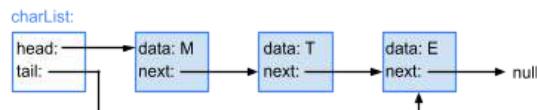
1. Appending an item to an empty list updates both the list's head and tail pointers.
2. Appending to a non-empty list adds the new node after the tail node and updates the tail pointer.

PARTICIPATION ACTIVITY

5.2.4: Appending a node to a singly-linked list.



- 1) Appending node D to charList updates which node's next pointer?



- M
- T
- E

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADO CSPB2270Summer2023

- 2) Appending node W to sampleList updates which of sampleList's pointers?



sampleList:

```
head: null
tail: null
```

head and tail

head only

tail only

- 3) Which statement is NOT executed when node 70 is appended to ticketList?

ticketList:



- list->head = newNode
- list->tail->next = newNode
- list->tail = newNode

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Prepending a node to a singly-linked list

Given a new node, the **Prepend** operation for a singly-linked list inserts the new node before the list's head node. The prepend algorithm behavior differs if the list is empty versus not empty:

- *Prepend to empty list:* If the list's head pointer is null (empty), the algorithm points the list's head and tail pointers to the new node.
- *Prepend to non-empty list:* If the list's head pointer is not null (not empty), the algorithm points the new node's next pointer to the head node, and then points the list's head pointer to the new node.

PARTICIPATION ACTIVITY

5.2.5: Singly-linked list: Prepending a node.



Animation content:

undefined

Animation captions:

1. Prepending an item to an empty list points the list's head and tail pointers to new node.
2. Prepending to a non-empty list points the new node's next pointer to the list's head node.
3. Prepending then points the list's head pointer to the new node.

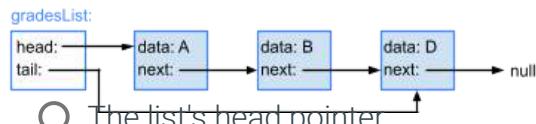
©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

5.2.6: Prepending a node in a singly-linked list.



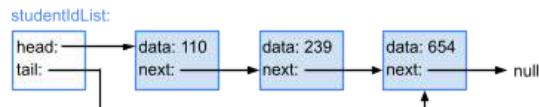
- 1) Prepending C to gradesList updates which pointer?



- The list's head pointer
- A's next pointer
- D's next pointer

2) Prepending node 789 to studentIdList
updates the list's tail pointer.

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



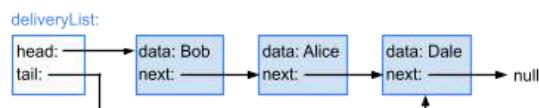
- True
- False

3) Prepending node 6 to parkingList
updates the list's tail pointer.



- True
- False

4) Prepending Evelyn to deliveryList
executes which statement?



- `list->head = null`
- `newNode->next = list->head`
- `list->tail = newNode`

CHALLENGE ACTIVITY

5.2.1: Singly-linked lists.

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

489394.3384924.qx3zqy7

Start

What is numList after the following operations?

```
numList = new List  
ListAppend(numList, node 73)  
ListAppend(numList, node 96)
```

numList is now: Ex: 1, 2, 3 (comma between values)

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSBP2270Summer2023



5.3 List data structure



This section has been set as optional by your instructor.

A common approach for implementing a linked list is using two data structures:

1. List data structure: A **list data structure** is a data structure containing the list's head and tail, and may also include additional information, such as the list's size.
2. List node data structure: The list node data structure maintains the data for each list element, including the element's data and pointers to the other list element.

A list data structure is not required to implement a linked list, but offers a convenient way to store the list's head and tail. When using a list data structure, functions that operate on a list can use a single parameter for the list's data structure to manage the list.

A linked list can also be implemented without using a list data structure, which minimally requires using separate list node pointer variables to keep track of the list's head.

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSBP2270Summer2023

PARTICIPATION ACTIVITY

5.3.1: Linked lists can be stored with or without a list data structure.



Animation content:

undefined

Animation captions:

1. A linked list can be maintained without a list data structure, but a pointer to the head and tail of the list must be stored elsewhere, often as local variables.
2. A list data structure stores both the head and tail pointers in one object.

PARTICIPATION ACTIVITY

5.3.2: Linked list data structure.

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 1) A linked list must have a list data structure.
 - True
 - False
- 2) A list data structure can have additional information besides the head and tail pointers.
 - True
 - False
- 3) A linked list has $O(n)$ space complexity, whether a list data structure is used or not.
 - True
 - False

5.4 Singly-linked lists: Insert

Given a new node, the **InsertAfter** operation for a singly-linked list inserts the new node after a provided existing list node. curNode is a pointer to an existing list node, but can be null when inserting into an empty list. The InsertAfter algorithm considers three insertion scenarios:

- *Insert as list's first node:* If the list's head pointer is null, the algorithm points the list's head and tail pointers to the new node.
- *Insert after list's tail node:* If the list's head pointer is not null (list not empty) and curNode points to the list's tail node, the algorithm points the tail node's next pointer and the list's tail pointer to the new node.
- *Insert in middle of list:* If the list's head pointer is not null (list not empty) and curNode does not point to the list's tail node, the algorithm points the new node's next pointer to curNode's next node, and then points curNode's next pointer to the new node.

**Animation content:**

undefined

Animation captions:

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

1. Inserting the list's first node points the list's head and tail pointers to newNode.
2. Inserting after the tail node points the tail node's next pointer to newNode.
3. Then, the list's tail pointer is pointed to newNode.
4. Inserting into the middle of the list points newNode's next pointer to curNode's next node.
5. Then, curNode's next pointer is pointed to newNode.



Type the list after the given operations. Type the list as: 5, 7, 9

1) numsList: 5, 9



ListInsertAfter(numsList, node 9,
node 4)

numsList:

//**Check****Show answer**

2) numsList: 23, 17, 8



ListInsertAfter(numsList, node 23,
node 5)

numsList:

//**Check****Show answer**

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

3) numsList: 1



ListInsertAfter(numsList, node 1,
node 6)
ListInsertAfter(numsList, node 1,
node 4)

numsList:

Check**Show answer**

- 4) numsList: 77



ListInsertAfter(numsList, node 77,
node 32)

ListInsertAfter(numsList, node 32,
node 50)

ListInsertAfter(numsList, node 32,
node 46)

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

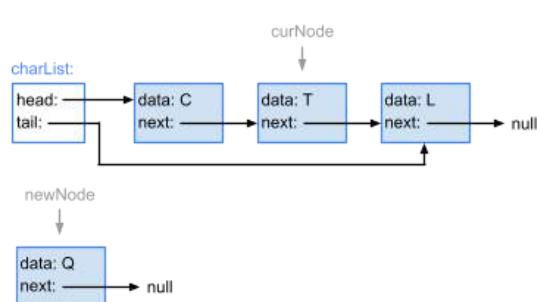
numsList:

Check**Show answer**
PARTICIPATION ACTIVITY

5.4.3: Singly-linked list insert-after algorithm.

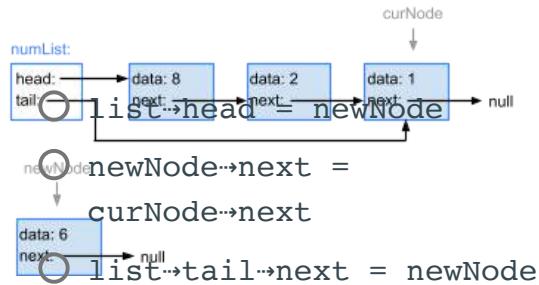


- 1) ListInsertAfter(charList, node T, node Q)
-
- assigns newNode's next pointer with

 curNode→next charList's head node null

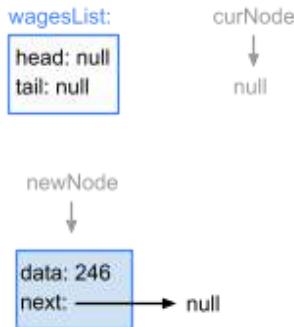
- 2) ListInsertAfter(numList, node 1, node 6)
-
- executes which statement?

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



- 3) ListInsertAfter(wagesList, list head, node 246) executes which statement?

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



- list→head = newNode
- list→tail→next = newNode
- curNode→next = newNode

CHALLENGE ACTIVITY

5.4.1: Singly-linked lists: Insert.



489394.3384924.qx3zqy7

Start

What is numList after the following operations?

numList: 62, 60

ListInsertAfter(numList, node 62, node 27)
ListInsertAfter(numList, node 27, node 95)
ListInsertAfter(numList, node 27, node 75)

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

numList is now: Ex: 1, 2, 3 (comma between values)

1

2

3

4

[Check](#)[Next](#)

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

5.5 Singly-linked lists: Remove

Given a specified existing node in a singly-linked list, the **RemoveAfter** operation removes the node after the specified list node. The existing node must be specified because each node in a singly-linked list only maintains a pointer to the next node.

The existing node is specified with the curNode parameter. If curNode is null, RemoveAfter removes the list's first node. Otherwise, the algorithm removes the node after curNode.

- *Remove list's head node (special case):* If curNode is null, the algorithm points sucNode to the head node's next node, and points the list's head pointer to sucNode. If sucNode is null, the only list node was removed, so the list's tail pointer is pointed to null (indicating the list is now empty).
- *Remove node after curNode:* If curNode's next pointer is not null (a node after curNode exists), the algorithm points sucNode to the node after curNode's next node. Then curNode's next pointer is pointed to sucNode. If sucNode is null, the list's tail node was removed, so the algorithm points the list's tail pointer to curNode (the new tail node).

PARTICIPATION ACTIVITY

5.5.1: Singly-linked list: Node removal.



Animation content:

undefined

Animation captions:

1. If curNode is null, the list's head node is removed.
2. The list's head pointer is pointed to the list head's successor node.
3. If node exists after curNode exists, that node is removed. sucNode points to node after the next node (i.e., the next next node).
4. curNode's next pointer is pointed to sucNode.
5. If sucNode is null, the list's tail node was removed. curNode is now the list tail node.
6. If list's tail node is removed, curNode's next pointer is null.
7. If list's tail node is removed, the list's tail pointer is pointed to curNode.

Taylor Larrechea
COLORADOCSPB2270Summer2023

**PARTICIPATION
ACTIVITY**

5.5.2: Removing nodes from a singly-linked list.

Type the list after the given operations. Type the list as: 4, 19, 3

1) numsList: 2, 5, 9



ListRemoveAfter(numsList, node 5)

numsList:

Check**Show answer**

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADO CSPB2270Summer2023

2) numsList: 3, 57, 28, 40



ListRemoveAfter(numsList, null)

numsList:

Check**Show answer**

3) numsList: 9, 4, 11, 7



ListRemoveAfter(numsList, node 11)

numsList:

Check**Show answer**

4) numsList: 10, 20, 30, 40, 50, 60



ListRemoveAfter(numsList, node 40)

ListRemoveAfter(numsList, node 20)

numsList:

Check**Show answer**

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADO CSPB2270Summer2023

5) numsList: 91, 80, 77, 60, 75



ListRemoveAfter(numsList, node

60)
ListRemoveAfter(numsList, node
77)
ListRemoveAfter(numsList, null)

numsList:

Check**Show answer**

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

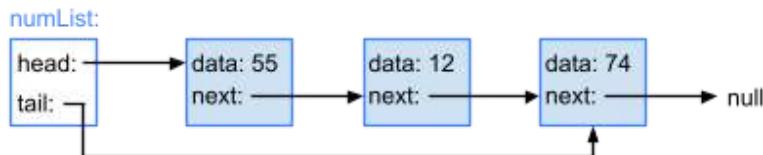
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

5.5.3: ListRemoveAfter algorithm execution: Intermediate node.



Given numList, ListRemoveAfter(numList, node 55) executes which of the following statements?



1) sucNode = list → head → next



- Yes
- No

2) curNode → next = sucNode



- Yes
- No

3) list → head = sucNode



- Yes
- No

4) list → tail = curNode



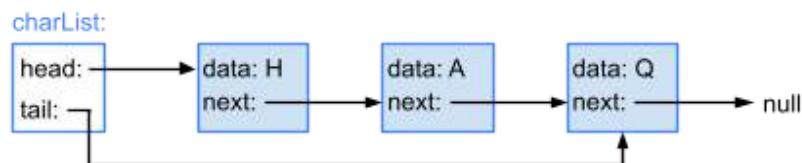
- Yes
- No

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023**PARTICIPATION ACTIVITY**

5.5.4: ListRemoveAfter algorithm execution: List head node.



Given `charList`, `ListRemoveAfter(charList, null)` executes which of the following statements?



©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADO CSPB2270Summer2023

1) `sucNode = list→head→next`

- Yes
- No

2) `curNode→next = sucNode`

- Yes
- No

3) `list→head = sucNode`

- Yes
- No

4) `list→tail = curNode`

- Yes
- No

CHALLENGE ACTIVITY

5.5.1: Singly-linked lists: Remove.

489394.3384924.qx3zqy7

Start

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADO CSPB2270Summer2023

Given list: 9, 5, 4, 1, 2

What list results from the following operations?

`ListRemoveAfter(list, node 1)`

`ListRemoveAfter(list, node 5)`

`ListRemoveAfter(list, null)`

List items in order, from head to tail.

Ex: 25, 42, 12

1

2

3

4

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSBP2270Summer2023

[Check](#)[Next](#)

5.6 Linked list search

Given a key, a **search** algorithm returns the first node whose data matches that key, or returns null if a matching node was not found. A simple linked list search algorithm checks the current node (initially the list's head node), returning that node if a match, else pointing the current node to the next node and repeating. If the pointer to the current node is null, the algorithm returns null (matching node was not found).

PARTICIPATION ACTIVITY

5.6.1: Singly-linked list: Searching.

**Animation content:**

undefined

Animation captions:

1. Search starts at list's head node. If node's data matches key, matching node is returned.
2. If no matching node is found, null is returned.

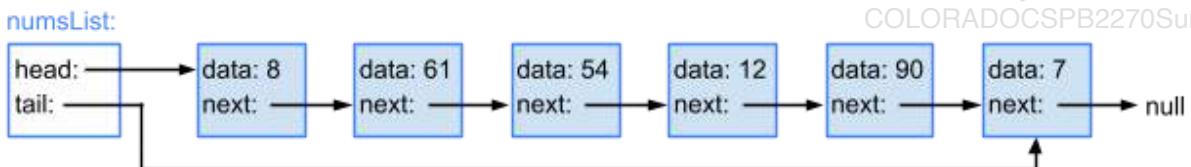
PARTICIPATION ACTIVITY

5.6.2: ListSearch algorithm execution.

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSBP2270Summer2023



- 1) How many nodes will ListSearch visit when searching for 54?



- 2) How many nodes will ListSearch visit when searching for 48?

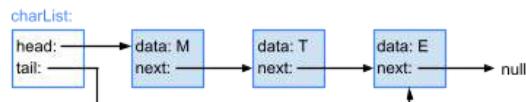
©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 3) What value does ListSearch return if the search key is not found?

PARTICIPATION ACTIVITY

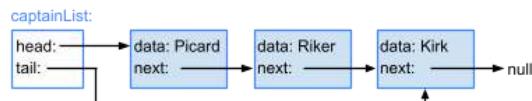
5.6.3: Searching a linked-list.

- 1) ListSearch(charList, E) first assigns curNode to ____.



- Node M
- Node T
- Node E

- 2) For ListSearch(captainList, Sisko), after checking node Riker, to which node is curNode pointed?



- node Riker
- node Kirk

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

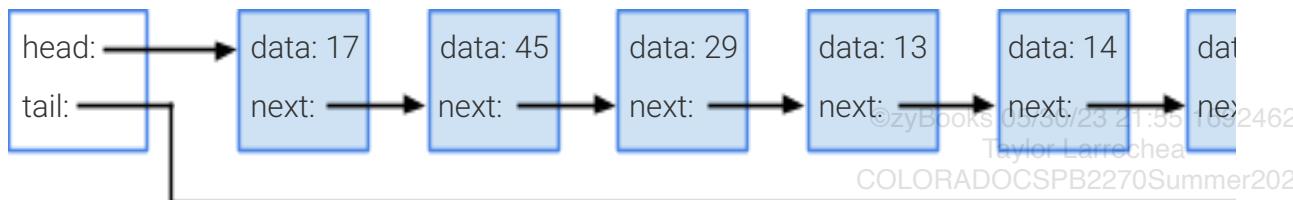
CHALLENGE ACTIVITY

5.6.1: Linked list search.

489394.3384924.qx3zqy7

Start

numList:



ListSearch(numList, 14) points the current pointer to node Ex: 9 after checking node 45.

ListSearch(numList, 14) will make Ex: 9 comparisons.

1

2

Check**Next**

5.7 Doubly-linked lists

Doubly-linked list

A **doubly-linked list** is a data structure for implementing a list ADT, where each node has data, a pointer to the next node, and a pointer to the previous node. The list structure typically points to the first node and the last node. The doubly-linked list's first node is called the head, and the last node the tail.

A doubly-linked list is similar to a singly-linked list, but instead of using a single pointer to the next node in the list, each node has a pointer to the next and previous nodes. Such a list is called "doubly-linked" because each node has two pointers, or "links". A doubly-linked list is a type of **positional list**: A list where elements contain pointers to the next and/or previous elements in the list.

PARTICIPATION ACTIVITY

5.7.1: Doubly-linked list data structure.

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADO CSPB2270Summer2023

- 1) Each node in a doubly-linked list contains data and ____ pointer(s).

one



two

- 2) Given a doubly-linked list with nodes 20, 67, 11, node 20 is the ____.

 head tail

- 3) Given a doubly-linked list with nodes 4, 7, 5, 1, node 7's previous pointer points to node ____.

 4 5

- 4) Given a doubly-linked list with nodes 8, 12, 7, 3, node 7's next pointer points to node ____.

 12 3

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Appending a node to a doubly-linked list

Given a new node, the **Append** operation for a doubly-linked list inserts the new node after the list's tail node. The append algorithm behavior differs if the list is empty versus not empty:

- *Append to empty list:* If the list's head pointer is null (empty), the algorithm points the list's head and tail pointers to the new node.
- *Append to non-empty list:* If the list's head pointer is not null (not empty), the algorithm points the tail node's next pointer to the new node, points the new node's previous pointer to the list's tail node, and points the list's tail pointer to the new node.

PARTICIPATION
ACTIVITY

5.7.2: Doubly-linked list: Appending a node.

Animation content:

undefined

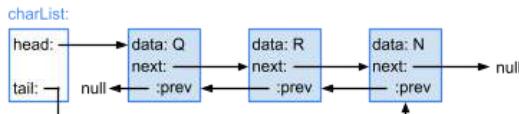
©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Animation captions:

1. Appending an item to an empty list updates the list's head and tail pointers.
2. Appending to a non-empty list adds the new node after the tail node and updates the tail pointer.
3. newNode's previous pointer is pointed to the list's tail node.
4. The list's tail pointer is then pointed to the new node.

PARTICIPATION ACTIVITY**5.7.3: Doubly-linked list data structure.**

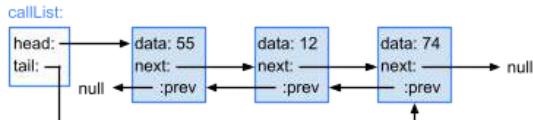
- 1) ListAppend(charList, node F) inserts node F ____.



©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- after node Q
- before node N
- after node N

- 2) ListAppend(callList, node 5) executes which statement?



- `list->head = newNode`
- `list->tail->next = newNode`
- `newNode->next = list->tail`

- 3) Appending node K to rentalList executes which of the following statements?



- `list->head = newNode`
- `list->tail->next = newNode`
- `newNode->prev = list->tail`

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Prepending a node to a doubly-linked list

Given a new node, the **Prepend** operation of a doubly-linked list inserts the new node before the list's head node and points the head pointer to the new node.

- *Prepend to empty list:* If the list's head pointer is null (empty), the algorithm points the list's head and tail pointers to the new node.

- *Prepend to non-empty list:* If the list's head pointer is not null (not empty), the algorithm points the new node's next pointer to the list's head node, points the list head node's previous pointer to the new node, and then points the list's head pointer to the new node.

PARTICIPATION ACTIVITY

5.7.4: Doubly-linked list: Prepending a node.

**Animation content:**

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSBP2270Summer2023

undefined

Animation captions:

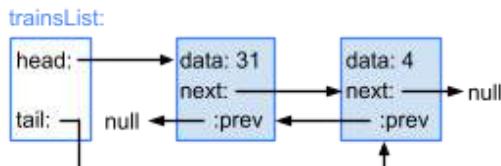
1. Prepending an item to an empty list points the list's head and tail pointers to new node.
2. Prepending to a non-empty list points new node's next pointer to the list's head node.
3. Prepending then points the head node's previous pointer to the new node.
4. Then the list's head pointer is pointed to the new node.

PARTICIPATION ACTIVITY

5.7.5: Prepending a node in a doubly-linked list.



- 1) Prepending 29 to trainsList updates the list's head pointer to point to node ____.



- 4
 29
 31

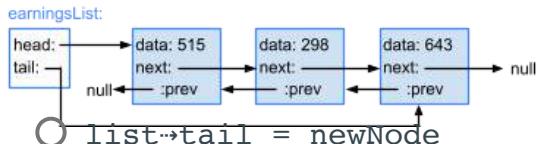
- 2) ListPrepend(shoppingList, node Milk) updates the list's tail pointer.

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSBP2270Summer2023

- True
 False

- 3) ListPrepend(earningsList, node 977) executes which statement?





- `list->tail = newNode`
- `newNode->next = list->head`
- `newNode->next = list->tail`

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

CHALLENGE ACTIVITY

5.7.1: Doubly-linked lists.



489394.3384924.qx3zqy7

Start

```

numList = new List
ListAppend(numList, node 31)
ListAppend(numList, node 98)
ListAppend(numList, node 13)
ListAppend(numList, node 55)
  
```

numList is now: Ex: 1, 2, 3 (comma between values)

Which node has a null next pointer? Ex: 5

Which node has a null previous pointer? Ex: 5

1

2

3

4

5

Check**Next**

5.8 Doubly-linked lists: Insert

Given a new node, the **InsertAfter** operation for a doubly-linked list inserts the new node after a provided existing list node. `curNode` is a pointer to an existing list node. The `InsertAfter` algorithm considers three insertion scenarios:

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

- *Insert as first node:* If the list's head pointer is null (list is empty), the algorithm points the list's head and tail pointers to the new node.
- *Insert after list's tail node:* If the list's head pointer is not null (list is not empty) and curNode points to the list's tail node, the new node is inserted after the tail node. The algorithm points the tail node's next pointer to the new node, points the new node's previous pointer to the list's tail node, and then points the list's tail pointer to the new node.
- *Insert in middle of list:* If the list's head pointer is not null (list is not empty) and curNode does not point to the list's tail node, the algorithm updates the current, new, and successor nodes' next and previous pointers to achieve the ordering {curNode newNode sucNode}, which requires four pointer updates: point the new node's next pointer to sucNode, point the new node's previous pointer to curNode, point curNode's next pointer to the new node, and point sucNode's previous pointer to the new node.

PARTICIPATION ACTIVITY

5.8.1: Doubly-linked list: Inserting nodes.

**Animation content:**

undefined

Animation captions:

1. Inserting a first node into the list points the list's head and tail pointers to the new node.
2. Inserting after the list's tail node points the tail node's next pointer to the new node.
3. Then the new node's previous pointer is pointed to the list's tail node. Finally, the list's tail pointer is pointed to the new node.
4. Inserting in the middle of a list points sucNode to curNode's successor (curNode's next node), then points newNode's next pointer to the successor node....
5. ...then points newNode's previous pointer to curNode...
6. ...and finally points curNode's next pointer to the new node.
7. Finally, points sucNode's previous pointer to the new node. At most, four pointers are updated to insert a new node in the list.

PARTICIPATION ACTIVITY

5.8.2: Inserting nodes in a doubly-linked list.



Given weeklySalesList: 12, 30

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Show the node order after the following operations:

ListInsertAfter(weeklySalesList, list tail, node 8)

ListInsertAfter(weeklySalesList, list head, node 45)

ListInsertAfter(weeklySalesList, node 45, node 76)

If unable to drag and drop, refresh the page.

node 76

node 45

node 30

node 8

node 12

Position 0 (list's head node)

Position 1

Position 2

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Position 3

Position 4 (list's tail node)

Reset

CHALLENGE ACTIVITY

5.8.1: Doubly-linked lists: Insert.



489394.3384924.qx3zqy7

Start

What is numList after the following operations?

numList: 87, 17

ListInsertAfter(numList, node 17, node 89)

ListInsertAfter(numList, node 89, node 50)

ListInsertAfter(numList, node 89, node 91)

numList is now: Ex: 1, 2, 3 (comma between values)

What node does node 91's next pointer point to?

What node does node 91's previous pointer point to?

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

1

2

3

4

Check

Next

5.9 Doubly-linked lists: Remove

The **Remove** operation for a doubly-linked list removes a provided existing list node. curNode is a pointer to an existing list node. The algorithm first determines the node's successor (the next node) and predecessor (the previous node). The variable sucNode points to the node's successor, and the variable predNode points to the node's predecessor. The algorithm uses four separate checks to update each pointer:

On Page 55 of 215 1699102

Taylor Larrechea

COLORADO CSPB2270Summer2023

- Successor exists: If the successor node pointer is not null (successor exists), the algorithm points the successor's previous pointer to the predecessor node.
- Predecessor exists: If the predecessor node pointer is not null (predecessor exists), the algorithm points the predecessor's next pointer to the successor node.
- Removing list's head node: If curNode points to the list's head node, the algorithm points the list's head pointer to the successor node.
- Removing list's tail node: If curNode points to the list's tail node, the algorithm points the list's tail pointer to the predecessor node.

When removing a node in the middle of the list, both the predecessor and successor nodes exist, and the algorithm updates the predecessor and successor nodes' pointers to achieve the ordering {predNode sucNode}. When removing the only node in a list, curNode points to both the list's head and tail nodes, and sucNode and predNode are both null. So, the algorithm points the list's head and tail pointers to null, making the list empty.

PARTICIPATION ACTIVITY

5.9.1: Doubly-linked list: Node removal.



Animation content:

undefined

Animation captions:

1. curNode points to the node to be removed. sucNode points to curNode's successor (curNode's next node). predNode points to curNode's predecessor (curNode's previous node).
2. sucNode's previous pointer is pointed to the node preceding curNode.
3. If curNode points to the list's head node, the list's head pointer is pointed to the successor node. With the pointers updated, curNode can be removed.
4. curNode points to node 5, which will be removed. sucNode points to node 2. predNode points to node 4.
5. The predecessor node's next pointer is pointed to the successor node. The successor node's previous pointer is pointed to the predecessor node. With pointers updated, curNode can be removed.
6. curNode points to node 2, which will be removed. sucNode points to nothing (null). predNode points to node 4.

On Page 462 of 215 1699102

Taylor Larrechea

COLORADO CSPB2270Summer2023

7. The predecessor node's next pointer is pointed to the successor node. If curNode points to the list's tail node, the list's tail pointer is assigned with predNode. With pointers updated, curNode can be removed.

PARTICIPATION ACTIVITY

5.9.2: Deleting nodes from a doubly-linked list.



Type the list after the given operations. Type the list as: 4, 19, 3

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

1) numsList: 71, 29, 54



ListRemove(numsList, node 29)

numsList:

Check**Show answer**

2) numsList: 2, 8, 1



ListRemove(numsList, list tail)

numsList:

Check**Show answer**

3) numsList: 70, 82, 41, 120, 357, 66



ListRemove(numsList, node 82)

ListRemove(numsList, node 357)

ListRemove(numsList, node 66)

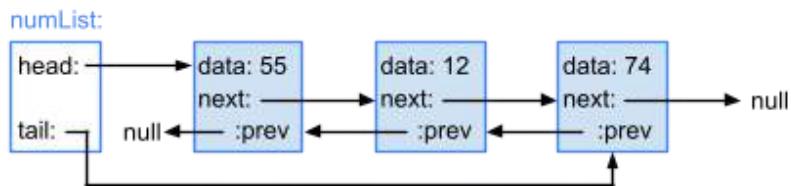
numsList:

Check**Show answer****PARTICIPATION ACTIVITY**

5.9.3: ListRemove algorithm execution: Intermediate node.



Given numList, ListRemove(numList, node 12) executes which of the following statements?



1) sucNode \rightarrow prev = predNode

- Yes
- No

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

2) predNode \rightarrow next = sucNode

- Yes
- No



3) list \rightarrow head = sucNode

- Yes
- No



4) list \rightarrow tail = predNode

- Yes
- No

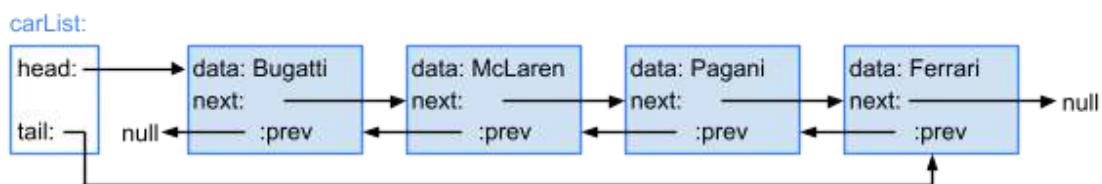


PARTICIPATION ACTIVITY

5.9.4: ListRemove algorithm execution: List head node.



Given carList, ListRemove(carList, node Bugatti) executes which of the following statements?



©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

1) sucNode \rightarrow prev = predNode

- Yes
- No



2) predNode \rightarrow next = sucNode

- Yes



No

3) list->head = sucNode

 Yes No

4) list->tail = predNode

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

 Yes No**CHALLENGE ACTIVITY**

5.9.1: Doubly-linked lists: Remove.



489394.3384924.qx3zqy7

Start

Given numList: 3, 5, 4, 1, 2, 8

What is numList after the following operations?

ListRemove(numList, node 8)

ListRemove(numList, node 3)

ListRemove(numList, node 1)

numList is now: Ex: 25, 42, 12

1

2

3

4

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Check**Next**

5.10 Linked list traversal

Linked list traversal

A **list traversal** algorithm visits all nodes in the list once and performs an operation on each node. A common traversal operation prints all list nodes. The algorithm starts by pointing a curNode pointer to the list's head node. While curNode is not null, the algorithm prints the current node, and then points curNode to the next node. After the list's tail node is visited, curNode is pointed to the tail node's next node, which does not exist. So, curNode is null, and the traversal ends. The traversal algorithm supports both singly-linked and doubly-linked lists.

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADO CSPB2270Summer2023

Figure 5.10.1: Linked list traversal algorithm.

```
ListTraverse(list) {  
    curNode = list->head // Start at  
    head  
  
    while (curNode is not null) {  
        Print curNode's data  
        curNode = curNode->next  
    }  
}
```

PARTICIPATION ACTIVITY

5.10.1: Singly-linked list: List traversal.



Animation content:

undefined

Animation captions:

1. Traverse starts at the list's head node.
2. curNode's data is printed, and then curNode is pointed to the next node.
3. After the list's tail node is printed, curNode is pointed to the tail node's next node, which does not exist.
4. The traversal ends when curNode is null.

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADO CSPB2270Summer2023

PARTICIPATION ACTIVITY

5.10.2: List traversal.



1) ListTraverse begins with ____.

- a specified list node
- the list's head node

- the list's tail node

- 2) Given numsList is: 5, 8, 2, 1.
ListTraverse(numsList) visits ____ node(s).

- one
- two
- four

- 3) ListTraverse can be used to traverse a doubly-linked list.

- True
- False

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Doubly-linked list reverse traversal

A doubly-linked list also supports a reverse traversal. A **reverse traversal** visits all nodes starting with the list's tail node and ending after visiting the list's head node.

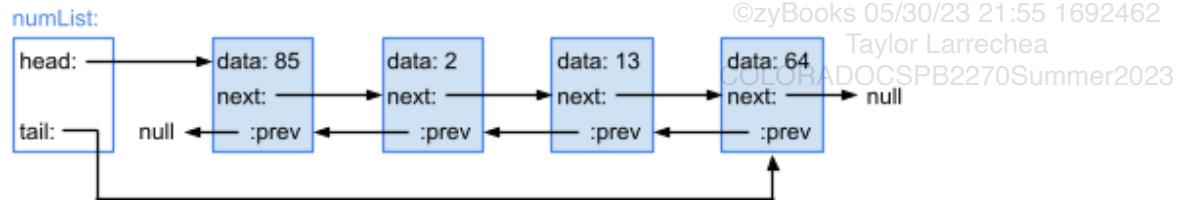
Figure 5.10.2: Reverse traversal algorithm.

```
ListTraverseReverse(list) {
    curNode = list->tail // Start at tail

    while (curNode is not null) {
        Print curNode's data
        curNode = curNode->prev
    }
}
```

PARTICIPATION ACTIVITY

5.10.3: Reverse traversal algorithm execution.



- 1) ListTraverseReverse visits which node second?
 Node 2

Node 13

- 2) ListTraverseReverse can be used to traverse a singly-linked list.

 True False

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

5.11 Sorting linked lists

Insertion sort for doubly-linked lists

Insertion sort for a doubly-linked list operates similarly to the insertion sort algorithm used for arrays. Starting with the second list element, each element in the linked list is visited. Each visited element is moved back as needed and inserted into the correct position in the list's sorted portion. The list must be a doubly-linked list, since backward traversal is not possible in a singly-linked list.

PARTICIPATION ACTIVITY

5.11.1: Sorting a doubly-linked list with insertion sort.



Animation content:

undefined

Animation captions:

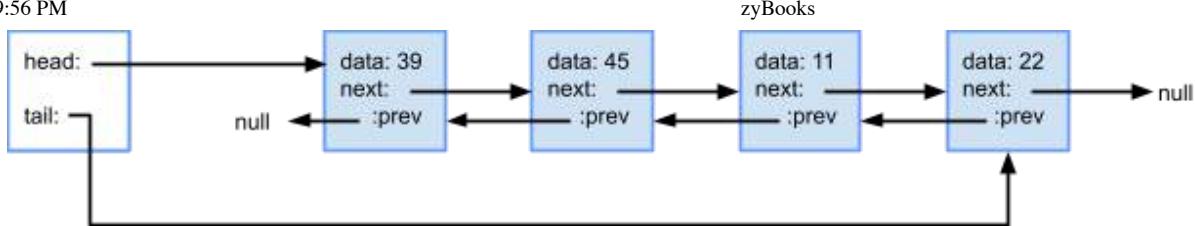
1. The curNode pointer begins at node 91 in the list.
2. searchNode starts at node 81 and does not move because 81 is not greater than 91. Removing and re-inserting node 91 after node 81 does not change the list.
3. For node 23, searchNode traverses the list backward until becoming null. Node 23 is prepended as the new list head.
4. Node 49 is inserted after node 23, using ListInsertAfter.
5. Node 12 is inserted before node 23, using ListPrepend, to complete the sort.

PARTICIPATION ACTIVITY

5.11.2: Insertion sort for doubly-linked lists.

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Suppose ListInsertionSortDoublyLinked is executed to sort the list below.



1) What is the first node that curNode will point to?

- Node 39
- Node 45
- Node 11



©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADO CSPB2270Summer2023

2) The ordering of list nodes is not altered when node 45 is removed and then inserted after node 39.

- True
- False



3) ListPrepend is called on which node(s)?

- Node 11 only
- Node 22 only
- Nodes 11 and 22



Algorithm efficiency

Insertion sort's typical runtime is $O(N^2)$. If a list has N elements, the outer loop executes $N - 1$ times. For each outer loop execution, the inner loop may need to examine all elements in the sorted part. Thus, the inner loop executes on average times. So the total number of comparisons is proportional to , or $O(N^2)$. In the best case scenario, the list is already sorted, and the runtime complexity is $O(N)$.

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADO CSPB2270Summer2023

Insertion sort for singly-linked lists

Insertion sort can sort a singly-linked list by changing how each visited element is inserted into the sorted portion of the list. The standard insertion sort algorithm traverses the list from the current element toward the list head to find the insertion position. For a singly-linked list, the insertion sort

algorithm can find the insertion position by traversing the list from the list head toward the current element.

Since a singly-linked list only supports inserting a node after an existing list node, the ListFindInsertionPosition algorithm searches the list for the insertion position and returns the list node after which the current node should be inserted. If the current node should be inserted at the head, ListFindInsertionPosition returns null.

PARTICIPATION ACTIVITY**5.11.3: Sorting a singly-linked list with insertion sort.**

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Animation content:

undefined

Animation captions:

1. Insertion sort for a singly-linked list initializes curNode to point to the second list element, or node 56.
2. ListFindInsertionPosition searches the list from the head toward the current node to find the insertion position. ListFindInsertionPosition returns null, so Node 56 is prepended as the list head.
3. Node 64 is less than node 71. ListFindInsertionPosition returns a pointer to the node before node 71, or node 56. Then, node 64 is inserted after node 56.
4. The insertion position for node 87 is after node 71. Node 87 is already in the correct position and is not moved.
5. Although node 74 is only moved back one position, ListFindInsertionPosition compared node 74 with all other nodes' values to find the insertion position.

Figure 5.11.1: ListFindInsertionPosition algorithm.

```
ListFindInsertionPosition(list, dataValue) {  
    curNodeA = null  
    curNodeB = list->head  
    while (curNodeB != null and dataValue >  
curNodeB->data) {  
        curNodeA = curNodeB  
        curNodeB = curNodeB->next  
    }  
    return curNodeA  
}
```

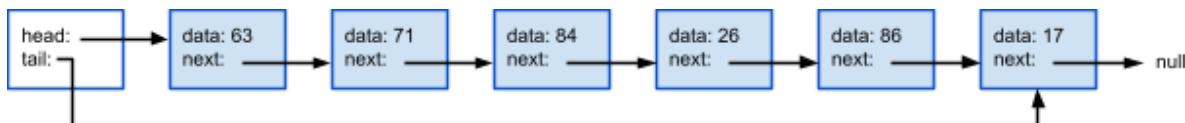
©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY**5.11.4: Sorting singly-linked lists with insertion sort.**

Given `ListInsertionSortSinglyLinked` is called to sort the list below.



- 1) What is returned by the first call to `ListFindInsertPosition`?

- null
- Node 63
- Node 71
- Node 84

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 2) How many times is `ListPrepend` called?

- 0
- 1
- 2



- 3) How many times is `ListInsertAfter` called?

- 0
- 1
- 2



Algorithm efficiency

The average and worst case runtime of `ListInsertionSortSinglyLinked` is $O(n^2)$. The best case runtime is $O(n)$, which occurs when the list is sorted in descending order.

Sorting linked-lists vs. arrays

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Sorting algorithms for arrays, such as quicksort and heapsort, require constant-time access to arbitrary, indexed locations to operate efficiently. Linked lists do not allow indexed access, making it difficult to adapt such sorting algorithms to operate on linked lists. The tables below provide a brief overview of the challenges in adapting array sorting algorithms for linked lists.

Table 5.11.1: Sorting algorithms easily adapted to efficiently sort linked lists.

Sorting algorithm	Adaptation to linked lists
Insertion sort	Operates similarly on doubly-linked lists. Requires searching from the head of the list for an element's insertion position for singly-linked lists.
Merge sort	Finding the middle of the list requires searching linearly from the head of the list. The merge algorithm can also merge lists without additional storage.

Table 5.11.2: Sorting algorithms difficult to adapt to efficiently sort linked lists.

Sorting algorithm	Challenge
Shell sort	Jumping the gap between elements cannot be done on a linked list, as each element between two elements must be traversed.
Quicksort	Partitioning requires backward traversal through the right portion of the array. Singly-linked lists do not support backward traversal.
Heap sort	Indexed access is required to find child nodes in constant time when percolating down.

PARTICIPATION ACTIVITY

5.11.5: Sorting linked-lists vs. sorting arrays.

- What aspect of linked lists makes adapting array-based sorting algorithms to linked lists difficult?

- Two elements in a linked list cannot be swapped in constant time.
- Nodes in a linked list cannot be moved.

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- Elements in a linked list cannot be accessed by index.
- 2) Which sorting algorithm uses a gap value to jump between elements, and is difficult to adapt to linked lists for this reason?



- Insertion sort
- Merge sort
- Shell sort

- 3) Why are sorting algorithms for arrays generally more difficult to adapt to singly-linked lists than to doubly-linked lists?



- Singly-linked lists do not support backward traversal.
- Singly-linked lists do not support inserting nodes at arbitrary locations.

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

5.12 Linked list dummy nodes

Dummy nodes

A linked list implementation may use a **dummy node** (or **header node**): A node with an unused data member that always resides at the head of the list and cannot be removed. Using a dummy node simplifies the algorithms for a linked list because the head and tail pointers are never null.

An empty list consists of the dummy node, which has the next pointer set to null, and the list's head and tail pointers both point to the dummy node.

PARTICIPATION ACTIVITY

5.12.1: Singly-linked lists with and without a dummy node

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Animation captions:

1. An empty linked list without a dummy node has null head and tail pointers.
2. An empty linked list with a dummy node has the head and tail pointing to a node with null data.
3. Without the dummy node, a non-empty list's head pointer points to the first list item.

4. With a dummy node, the list's head pointer always points to the dummy node. The dummy node's next pointer points to the first list item.

PARTICIPATION ACTIVITY**5.12.2: Singly linked lists with a dummy node.**

- 1) The head and tail pointers always point to the dummy node.

- True
- False

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 2) The dummy node's next pointer points to the first list item.

- True
- False

**PARTICIPATION ACTIVITY****5.12.3: Condition for an empty list.**

- 1) If myList is a singly-linked list with a dummy node, which statement is true when the list is empty?

- `myList->head == null`
- `myList->tail == null`
- `myList->head == myList->tail`



Singly-linked list implementation

When a singly-linked list with a dummy node is created, the dummy node is allocated and the list's head and tail pointers are set to point to the dummy node.

List operations such as append, prepend, insert after, and remove after are simpler to implement compared to a linked list without a dummy node, since a special case is removed from each implementation. ListAppend, ListPrepend, and ListInsertAfter do not need to check if the list's head is null, since the list's head will always point to the dummy node. ListRemoveAfter does not need a special case to allow removal of the first list item, since the first list item is after the dummy node.

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Figure 5.12.1: Singly-linked list with dummy node: append, prepend, insert after, and remove after operations.

```
ListAppend(list, newNode) {  
    list->tail->next = newNode  
    list->tail = newNode  
}  
  
ListPrepend(list, newNode) {  
    newNode->next = list->head->next  
    list->head->next = newNode  
    if (list->head == list->tail) { // empty list  
        list->tail = newNode;  
    }  
}  
  
ListInsertAfter(list, curNode, newNode) {  
    if (curNode == list->tail) { // Insert after tail  
        list->tail->next = newNode  
        list->tail = newNode  
    }  
    else {  
        newNode->next = curNode->next  
        curNode->next = newNode  
    }  
}  
  
ListRemoveAfter(list, curNode) {  
    if (curNode is not null and curNode->next is not  
null) {  
        sucNode = curNode->next->next  
        curNode->next = sucNode  
  
        if (sucNode is null) {  
            // Removed tail  
            list->tail = curNode  
        }  
    }  
}
```

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY**5.12.4: Singly-linked list with dummy node.**

Suppose dataList is a singly-linked list with a dummy node.

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 1) Which statement removes the first item from the list?

- `ListRemoveAfter(dataList,
null)`
- `ListRemoveAfter(dataList,
dataList->head)`

- ListRemoveAfter(dataList,
 dataList->tail)

2) Which is a requirement of the ListPrepend function?

- The list is empty
- The list is not empty
- newNode is not null

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADO CSPB2270Summer2023

**PARTICIPATION
ACTIVITY**

5.12.5: Singly-linked list with dummy node.

Suppose numbersList is a singly-linked list with items 73, 19, and 86. Item 86 is at the list's tail.

1) What is the list's contents after the following operations?

```
lastItem = numbersList->tail
ListAppend(numbersList, node
25)
ListInsertAfter(numbersList,
lastItem, node 49)
```

- 73, 19, 86, 25, 49
- 73, 19, 86, 49, 25
- 73, 19, 25, 49, 86

2) Suppose the following statement is executed:

```
node19 =
numbersList->head->next->next
```

Which subsequent operations swap nodes 73 and 19?

- ListPrepend(numbersList,
node19)
- ListInsertAfter(numbersList,
numbersList->head, node19)
- ListRemoveAfter(numbersList,
numbersList->head->next)
ListPrepend(numbersList,
node19)

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADO CSPB2270Summer2023

Doubly-linked list implementation

A dummy node can also be used in a doubly-linked list implementation. The dummy node in a doubly-linked list always has the prev pointer set to null. ListRemove's implementation does not allow removal of the dummy node.

Figure 5.12.2: Doubly-linked list with dummy node: append, prepend, insert after, and remove operations.

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSBP2270Summer2023

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSBP2270Summer2023

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

```
ListAppend(list, newNode) {
    list->tail->next = newNode
    newNode->prev = list->tail
    list->tail = newNode
}

ListPrepend(list, newNode) {
    firstNode = list->head->next
    // Set the next and prev pointers for newNode
    newNode->next = list->head->next
    newNode->prev = list->head

    // Set the dummy node's next pointer
    list->head->next = newNode

    // Set prev on former first node
    if (firstNode is not null) {
        firstNode->prev = newNode
    }
}

ListInsertAfter(list, curNode, newNode) {
    if (curNode == list->tail) { // Insert after tail
        list->tail->next = newNode
        newNode->prev = list->tail
        list->tail = newNode
    }
    else {
        sucNode = curNode->next
        newNode->next = sucNode
        newNode->prev = curNode
        curNode->next = newNode
        sucNode->prev = newNode
    }
}

ListRemove(list, curNode) {
    if (curNode == list->head) {
        // Dummy node cannot be removed
        return
    }

    sucNode = curNode->next
    predNode = curNode->prev

    if (sucNode is not null) {
        sucNode->prev = predNode
    }

    // Predecessor node is always non-null
    predNode->next = sucNode

    if (curNode == list->tail) { // Removed tail
        list->tail = predNode
    }
}
```

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea
COLORADO CSPB2270Summer2023

PARTICIPATION ACTIVITY

5.12.6: Doubly-linked list with dummy node.



- 1) `ListPrepend(list, newNode)` is equivalent to

```
ListInsertAfter(list,  
list->head, newNode).
```

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- True
 False

- 2) ListRemove's implementation must not allow removal of the dummy node.

- True
 False

- 3) `ListInsertAfter(list, null, newNode)` will insert newNode before the list's dummy node.

- True
 False

Dummy head and tail nodes

A doubly-linked list implementation can also use 2 dummy nodes: one at the head and the other at the tail. Doing so removes additional conditionals and further simplifies the implementation of most methods.

PARTICIPATION ACTIVITY

5.12.7: Doubly-linked list append and prepend with 2 dummy nodes.



Animation content:

undefined

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Animation captions:

1. A list with 2 dummy nodes is initialized such that the list's head and tail point to 2 distinct nodes. Data is null for both nodes.
2. Prepending inserts after the head. The list head's next pointer is never null, even when the list is empty, because of the dummy node at the tail.
3. Appending inserts before the tail, since the list's tail pointer always points to the dummy node.

Figure 5.12.3: Doubly-linked list with 2 dummy nodes: insert after and remove operations.

```
ListInsertAfter(list, curNode, newNode) {
    if (curNode == list->tail) {
        // Can't insert after dummy tail
        return
    }

    sucNode = curNode->next
    newNode->next = sucNode
    newNode->prev = curNode
    curNode->next = newNode
    sucNode->prev = newNode
}

ListRemove(list, curNode) {
    if (curNode == list->head || curNode ==
list->tail) {
        // Dummy nodes cannot be removed
        return
    }

    sucNode = curNode->next
    predNode = curNode->prev

    // Successor node is never null
    sucNode->prev = predNode

    // Predecessor node is never null
    predNode->next = sucNode
}
```

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Removing if statements from ListInsertAfter and ListRemove

The if statement at the beginning of ListInsertAfter may be removed in favor of having a precondition that curNode cannot point to the dummy tail node. Likewise, ListRemove can remove the if statement and have a precondition that curNode cannot point to either dummy node. If such preconditions are met, neither function requires any if statements.

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

5.12.8: Comparing a doubly-linked list with 1 dummy node vs. 2 dummy nodes.



For each question, assume 2 list types are available: a doubly-linked list with 1 dummy node at the list's head, and a doubly-linked list with 2 dummy nodes, one at the head and the other at the tail.

1) When `list->head == list->tail` is true in

_____, the list is empty.

- a list with 1 dummy node
- a list with 2 dummy nodes
- either a list with 1 dummy node or a list with 2 dummy nodes

2) `list->tail` may be null in _____. □

- a list with 1 dummy node
- a list with 2 dummy nodes
- neither list type

3) `list->head->next` is always non-null in

_____.

- a list with 1 dummy node
- a list with 2 dummy nodes
- neither list type

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea
COLORADOCSPB2270Summer2023

5.13 Linked lists: Recursion

Forward traversal

Forward traversal through a linked list can be implemented using a recursive function that takes a node as an argument. If non-null, the node is visited first. Then, a recursive call is made on the node's next pointer, to traverse the remainder of the list.

The `ListTraverse` function takes a list as an argument, and searches the entire list by calling `ListTraverseRecursive` on the list's head.

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

5.13.1: Recursive forward traversal.

Animation content:

undefined

Animation captions:

1. ListTraverse begins traversal by calling the recursive function, ListTraverseRecursive, on the list's head.
2. The recursive function visits the node and calls itself for the next node.
3. Nodes 19 is visited and an additional recursive call visits node 41. The last recursive call encounters a null node and stops.

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

5.13.2: Forward traversal in a linked list with 10 nodes.



- 1) If ListTraverse is called to traverse a list with 10 nodes, how many calls to ListTraverseRecursive are made?

- 9
- 10
- 11

**PARTICIPATION ACTIVITY**

5.13.3: Forward traversal concepts.



- 1) ListTraverseRecursive works for both singly-linked and doubly-linked lists.

- True
- False



- 2) ListTraverseRecursive works for an empty list.

- True
- False



Searching

A recursive linked list search is implemented similar to forward traversal. Each call examines 1 node. If the node is null, then null is returned. Otherwise, the node's data is compared to the search key. If a match occurs, the node is returned, otherwise the remainder of the list is searched recursively.

Figure 5.13.1: ListSearch and ListSearchRecursive functions.

```
ListSearch(list, key) {  
    return ListSearchRecursive(key, list->head)  
}  
  
ListSearchRecursive(key, node) {  
    if (node is not null) {  
        if (node->data == key) {  
            return node  
        }  
        return ListSearchRecursive(key,  
node->next)  
    }  
    return null  
}
```

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

5.13.4: Searching a linked list with 10 nodes.



Suppose a linked list has 10 nodes.

- 1) When more than 1 of the list's nodes contains the search key, ListSearch returns ____ node containing the key.

- the first
- the last
- a random



- 2) Calling ListSearch results in a minimum of ____ calls to ListSearchRecursive.

- 1
- 2
- 10
- 11



- 3) When the key is not found, ListSearch returns ____.

- the list's head
- the list's tail
- null



©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Reverse traversal

Forward traversal visits a node first, then recursively traverses the remainder of the list. If the order is swapped, such that the recursive call is made first, the list is traversed in reverse order.

PARTICIPATION ACTIVITY
5.13.5: Recursive reverse traversal.

Animation content:

undefined

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

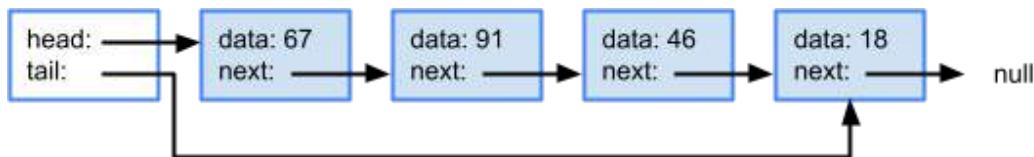
COLORADOCSPB2270Summer2023

Animation captions:

1. ListTraverseReverse is called to traverse the list. Much like a forward traversal, ListTraverseReverseRecursive is called for the list's head.
2. The recursive call on node 19 is made before visiting node 23.
3. Similarly, the recursive call on node 41 is made before visiting node 19, and the recursive call on null is made before visiting node 41.
4. The recursive call with the null node argument takes no action and is the first to return.
5. Execution returns to the line after the ListTraverseReverseRecursive(null) call. The node argument then points to node 41, which is the first node visited.
6. As the recursive calls complete, the remaining nodes are visited in reverse order. The last ListTraverseReverseRecursive call returns to ListTraverseReverse.
7. The entire list has been visited in reverse order.

PARTICIPATION ACTIVITY
5.13.6: Reverse traversal concepts.


Suppose ListTraverseReverse is called on the following list.



- 1) ListTraverseReverse passes _____ as the argument to ListTraverseReverseRecursive.

- node 67
- node 18
- null

- 2) ListTraverseReverseRecursive has been called for each of the list's nodes by the time the tail node is visited.

- True

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

False

- 3) If ListTraverseReverseRecursive were called directly on node 91, the nodes visited would be: _____.

- node 91 and node 67
- node 18, node 46, and node 91
- node 18, node 46, node 91, and node 67

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

5.14 Circular lists



This section has been set as optional by your instructor.

A **circular linked list** is a linked list where the tail node's next pointer points to the head of the list, instead of null. A circular linked list can be used to represent repeating processes. Ex: Ocean water evaporates, forms clouds, rains down on land, and flows through rivers back into the ocean. The head of a circular linked list is often referred to as the *start node*.

A traversal through a circular linked list is similar to traversal through a standard linked list, but must terminate after reaching the head node a second time, as opposed to terminating when reaching null.

PARTICIPATION ACTIVITY

5.14.1: Circular list structure and traversal.

**Animation content:**

undefined

Animation captions:

1. In a circular linked list, the tail node's next pointer points to the head node.
2. In a circular doubly-linked list, the head node's previous pointer points to the tail node.
3. Instead of stopping when the "current" pointer is null, traversal through a circular list stops when current comes back to the head node.

PARTICIPATION ACTIVITY

5.14.2: Circular list concepts.



- 1) Only a doubly-linked list can be circular.



True False

- 2) In a circular doubly-linked list with at least 2 nodes, where does the head node's previous pointer point to?

 List head List tail null

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

- 3) In a circular linked list with at least 2 nodes, where does the tail node's next pointer point to?

 List head List tail null

- 4) In a circular linked list with 1 node, the tail node's next pointer points to the tail.

 True False

- 5) The following code can be used to traverse a circular, doubly-linked list in reverse order.



```
CircularListTraverseReverse(tail)
{
    if (tail is not null) {
        current = tail
        do {
            visit current
            current =
current->previous
            } while (current != tail)
        }
}
```

 True False

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

5.15 Array-based lists



This section has been set as optional by your instructor.

Array-based lists

An **array-based list** is a list ADT implemented using an array. An array-based list supports the common list ADT operations, such as append, prepend, insert after, remove, and search.

In many programming languages, arrays have a fixed size. An array-based list implementation will dynamically allocate the array as needed as the number of elements changes. Initially, the array-based list implementation allocates a fixed size array and uses a length variable to keep track of how many array elements are in use. The list starts with a default allocation size, greater than or equal to 1. A default size of 1 to 10 is common.

Given a new element, the **append** operation for an array-based list of length X inserts the new element at the end of the list, or at index X.

PARTICIPATION
ACTIVITY

5.15.1: Appending to array-based lists.



Animation captions:

1. An array of length 4 is initialized for an empty list. Variables store the allocation size of 4 and list length of 0.
2. Appending 45 uses the first entry in the array, and the length is incremented to 1.
3. Appending 84, 12, and 78 uses the remaining space in the array.

PARTICIPATION
ACTIVITY

5.15.2: Array-based lists.



- 1) The length of an array-based list equals the list's array allocation size.
 - True
 - False
- 2) 42 is appended to an array-based list with allocationSize = 8 and length = 4. Appending assigns the array at index ____ with 42.
 - 4
 - 8
- 3) An array-based list can have a default allocation size of 0.

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- True
- False

Resize operation

An array-based list must be resized if an item is added when the allocation size equals the list length. A new array is allocated with a length greater than the existing array. Allocating the new array with twice the current length is a common approach. The existing array elements are then copied to the new array, which becomes the list's storage array.

Because all existing elements must be copied from 1 array to another, the resize operation has a runtime complexity of $O(n)$.

PARTICIPATION ACTIVITY

5.15.3: Array-based list resize operation.



Animation content:

undefined

Animation captions:

1. The allocation size and length of the list are both 4. An append operation cannot add to the existing array.
2. To resize, a new array is allocated of size 8, and the existing elements are copied to the new array. The new array replaces the list's array.
3. 51 can now be appended to the array.

PARTICIPATION ACTIVITY

5.15.4: Array-based list resize operation.



Assume the following operations are executed on the list shown below:

ArrayListAppend(list, 98)

ArrayListAppend(list, 42)

ArrayListAppend(list, 63)

array:

81	23	68	39	
----	----	----	----	--

allocationSize: 5

length : 4

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 1) Which operation causes ArrayListResize to be called?



- ArrayListAppend(list, 98)

- ArrayListAppend(list, 42)
 - ArrayListAppend(list, 63)
- 2) What is the list's length after 63 is appended?

- 5
- 7
- 10

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 3) What is the list's allocation size after 63 is appended?

- 5
- 7
- 10

Prepend and insert after operations

The **Prepend** operation for an array-based list inserts a new item at the start of the list. First, if the allocation size equals the list length, the array is resized. Then all existing array elements are moved up by 1 position, and the new item is inserted at the list start, or index 0. Because all existing array elements must be moved up by 1, the prepend operation has a runtime complexity of $O(n)$.

The **InsertAfter** operation for an array-based list inserts a new item after a specified index. Ex: If the contents of `numbersList` is: 5, 8, 2, `ArrayListInsertAfter(numbersList, 1, 7)` produces: 5, 8, 7, 2. First, if the allocation size equals the list length, the array is resized. Next, all elements in the array residing after the specified index are moved up by 1 position. Then, the new item is inserted at index (specified index + 1) in the list's array. The InsertAfter operation has a best case runtime complexity of $O(1)$ and a worst case runtime complexity of $O(n)$.

InsertAt operation.

Array-based lists often support the InsertAt operation, which inserts an item at a specified index. Inserting an item at a desired index X can be achieved by using `ArrayListInsertAfter(numbersList, X - 1, item)`.

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

5.15.5: Array-based list prepend and insert after operations.

Animation content:

Step 1: Data members for a list are shown: array, allocationSize, and length.

"array:" label is followed by 8 boxes for the array's data. Entries at indices 0 to 4 are: 45, 84, 12, 78, 51. Entries at indices 5, 6, and 7 are empty.

The other two labels are "allocationSize: 8" and "length: 5".

ArrayListPrepend(list, 91) begins execution. The first if statement's condition is false. Then the for loop executes, moving items up in the array, yielding: 45, 45, 84, 12, 78, 51. Array boxes for indices 6 and 7 remain empty.

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Step 2: Item 91 is assigned to index 0 in the array, yielding: 91, 45, 84, 12, 78, 51. Array boxes for indices 6 and 7 remain empty. Then length is incremented to 6.

Step 3: ArrayListInsertAfter(list, 2, 36) executes. The first if statement's condition is false, since $6 \neq 8$. The for loop moves items at indices 3, 4, and 5 up one, yielding: 91, 45, 84, 12, 12, 78, 51. Then 36 is assigned to index 3, yielding: 91, 45, 84, 36, 12, 78, 51. The array box for index 7 remains empty. Lastly, length is incremented to 7.

Animation captions:

1. To prepend 91, every array element is first moved up one index.
2. Item 91 is assigned to index 0 and length is incremented to 6.
3. Inserting item 36 after index 2 requires elements at indices 3 and higher to be moved up 1. Item 36 is inserted at index 3.

PARTICIPATION ACTIVITY

5.15.6: Array-based list prepend and insert after operations.



Assume the following operations are executed on the list shown below:

ArrayListPrepend(list, 76)

ArrayListInsertAfter(list, 1, 38)

ArrayListInsertAfter(list, 3, 91)

array:

22	16		
----	----	--	--

allocationSize: 4

length : 2

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

- 1) Which operation causes ArrayListResize to be called?

- ArrayListPrepend(list, 76)
- ArrayListInsertAfter(list, 1, 38)
- ArrayListInsertAfter(list, 3, 91)



- 2) What is the list's allocation size after all operations have completed?

- 5
- 8
- 10

- 3) What are the list's contents after all operations have completed?

- 22, 16, 76, 38, 91
- 76, 38, 22, 91, 16
- 76, 22, 38, 16, 91

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADO CSPB2270Summer2023

Search and removal operations

Given a key, the **search** operation returns the index for the first element whose data matches that key, or -1 if not found.

Given the index of an item in an array-based list, the **remove-at** operation removes the item at that index. When removing an item at index X, each item after index X is moved down by 1 position.

Both the search and remove operations have a worst case runtime complexity of O().

PARTICIPATION
ACTIVITY

5.15.7: Array-based list search and remove-at operations.



Animation content:

undefined

Animation captions:

1. The search for 84 compares against 3 elements before returning 2.
2. Removing the element at index 1 causes all elements after index 1 to be moved down to a lower index.
3. Decreasing the length by 1 effectively removes the last 51.
4. The search for 84 now returns 1.

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADO CSPB2270Summer2023

PARTICIPATION
ACTIVITY

5.15.8: Search and remove-at operations.



array:

94	82	16	48	26	45
----	----	----	----	----	----

allocationSize: 6

length : 6

- 1) What is the return value from
ArrayListSearch(list, 33)?

Check**Show answer**

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

- 2) When searching for 48, how many
elements in the list will be compared
with 48?

Check**Show answer**

- 3) ArrayListRemoveAt(list, 3) causes
how many items to be moved down
by 1 index?

Check**Show answer**

- 4) ArrayListRemoveAt(list, 5) causes
how many items to be moved down
by 1 index?

Check**Show answer****PARTICIPATION
ACTIVITY**

5.15.9: Search and remove-at operations.

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 1) Removing at index 0 yields the best
case runtime for remove-at.

 True False



2) Searching for a key that is not in the list yields the worst case runtime for search.

- True
- False

3) Neither search nor remove-at will resize the list's array.

- True
- False

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

CHALLENGE ACTIVITY

5.15.1: Array-based lists.



489394.3384924.qx3zqy7

Start

numList:

23	26		
----	----	--	--

 allocationSize: 4
length: 2

Determine the length and allocation size of numList after each operation. If an item is added w the allocation size equals the array length, a new array with twice the current length is allocate

Operation	Length	Allocation size
ArrayListAppend(numList, 22)	Ex: 1	Ex:1
ArrayListAppend(numList, 69)		
ArrayListAppend(numList, 36)		
ArrayListAppend(numList, 51)		
ArrayListAppend(numList, 39)		

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

1

2

3

4

Check**Next**

5.16 Stack abstract data type (ADT)



This section has been set as optional by your instructor.

Stack abstract data type

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSBP2270Summer2023

A **stack** is an ADT in which items are only inserted on or removed from the top of a stack. The stack **push** operation inserts an item on the top of the stack. The stack **pop** operation removes and returns the item at the top of the stack. Ex: After the operations "Push 7", "Push 14", "Push 9", and "Push 5", "Pop" returns 5. A second "Pop" returns 9. A stack is referred to as a **last-in first-out** ADT. A stack can be implemented using a linked list, an array, or a vector.

PARTICIPATION
ACTIVITY

5.16.1: Stack ADT.



Animation captions:

1. A new stack named "route" is created. Items can be pushed on the top of the stack.
2. Popping an item removes and returns the item from the top of the stack.

PARTICIPATION
ACTIVITY

5.16.2: Stack ADT: Push and pop operations.



- 1) Given numStack: 7, 5 (top is 7).

Type the stack after the following push operation. Type the stack as: 1, 2, 3



Push(numStack, 8)

Check

[Show answer](#)

- 2) Given numStack: 34, 20 (top is 34)

Type the stack after the following two push operations. Type the stack as: 1, 2, 3

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSBP2270Summer2023



Push(numStack, 11)

Push(numStack, 4)

Check**Show answer**

- 3) Given numStack: 5, 9, 1 (top is 5)
What is returned by the following
pop operation?



Pop(numStack)

Check**Show answer**

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

- 4) Given numStack: 5, 9, 1 (top is 5)
What is the stack after the following
pop operation? Type the stack as: 1,
2, 3



Pop(numStack)

Check**Show answer**

- 5) Given numStack: 2, 9, 5, 8, 1, 3 (top
is 2).
What is returned by the second pop
operation?



Pop(numStack)

Pop(numStack)

Check**Show answer**

- 6) Given numStack: 41, 8 (top is 41)
What is the stack after the following
operations? Type the stack as: 1, 2, 3



Pop(numStack)

Push(numStack, 2)

Push(numStack, 15)

Pop(numStack)

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

 Check

Show answer

Common stack ADT operations

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Table 5.16.1: Common stack ADT operations.

Operation	Description	Example starting with stack: 99, 77 (top is 99).
Push(stack, x)	Inserts x on top of stack	Push(stack, 44). Stack: 44, 99, 77
Pop(stack)	Returns and removes item at top of stack	Pop(stack) returns: 99. Stack: 77
Peek(stack)	Returns but does not remove item at top of stack	Peek(stack) returns 99. Stack still: 99, 77
IsEmpty(stack)	Returns true if stack has no items	IsEmpty(stack) returns false.
GetLength(stack)	Returns the number of items in the stack	GetLength(stack) returns 2.

Note: Pop and Peek operations should not be applied to an empty stack; the resulting behavior may be undefined.

PARTICIPATION ACTIVITY

5.16.3: Common stack ADT operations.



- 1) Given inventoryStack: 70, 888, -3, 2

What does GetLength(inventoryStack) return?

- 4
- 70

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 2) Given callStack: 2, 9, 4

What are the contents of the stack after Peek(callStack)?

- 2, 9, 4
- 9, 4





3) Given callStack: 2, 9, 4

What are the contents of the stack after
Pop(callStack)?

- 2, 9, 4
- 9, 4

4) Which operation determines if the stack
contains no items?

- Peek
- IsEmpty

5) Which operation should usually be
preceded by a check that the stack is
not empty?

- Pop
- Push

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



**CHALLENGE
ACTIVITY**

5.16.1: Stack ADT.



489394.3384924.qx3zqy7

Start

Given numStack: 43, 98 (top is 43)

What is the stack after the operations?

Pop(numStack)
Push(numStack, 87)

Ex: 1, 2, 3

After the above operations, what does GetLength(numStack) return?

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Ex: 5

1

2

3

[Check](#)[Next](#)

5.17 Stacks using linked lists

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



This section has been set as optional by your instructor.

A stack is often implemented using a linked list, with the list's head node being the stack's top. A push is performed by creating a new list node, assigning the node's data with the item, and prepending the node to the list. A pop is performed by assigning a local variable with the head node's data, removing the head node from the list, and then returning the local variable.

PARTICIPATION ACTIVITY

5.17.1: Stack implementation using a linked list.

**Animation content:**

undefined

Animation captions:

1. Pushing 45 onto the stack allocates a new node and prepends the node to the list.
2. Each push prepends a new node to the list.
3. A pop assigns a local variable with the list's head node's data, removes the head node, and returns the local variable.

PARTICIPATION ACTIVITY

5.17.2: Stack push and pop operations with a linked list.



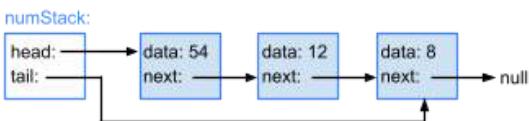
Assume the stack is implemented using a linked list.

- 1) An empty stack is indicated by a list head pointer value of ____.
 - newNode
 - null
 - Unknown
- 2) For StackPush(numStack, item 3),
newNode's next pointer is pointed to

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



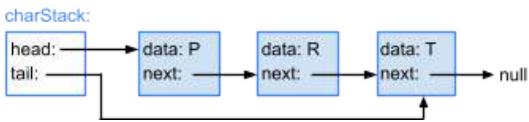
newNode

data: 3
next: null

○ Node 12

○ null

- 3) The operation StackPop(charStack) will remove which node?



○ Node P

○ Node R

○ Node T

- 4) StackPop returns list's head node.

True

False

CHALLENGE ACTIVITY

5.17.1: Stacks using linked lists.

489394.3384924.qx3zqy7

Start

Given an empty stack numStack,

What does the list head pointer point to? If the pointer is null, enter null.

Ex: 5 or null

After the operations, which node does the list head pointer point to?

```
StackPush(numStack, 17)  
StackPush(numStack, 73)  
StackPush(numStack, 57)
```

Ex: 5 or null

1

2

3

[Check](#)[Next](#)

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

5.18 Array-based stacks

Array-based stack storage

A stack can be implemented with an array. Two variables are needed in addition to the array:

- allocationSize: an integer for the array's allocated size.
- length: an integer for the stack's length.

The stack's bottom item is at `array[0]` and the top item is at `array[length - 1]`. If the stack is empty, length is 0.

PARTICIPATION ACTIVITY

5.18.1: Array-based stack storage.



Animation content:

undefined

Animation captions:

1. Stack 1's allocationSize and length are both 4. Array element 0 is at the stack's bottom. Array element 3 is at the stack's top.
2. Stack 2's length, 2, is less than allocationSize, 4. So elements at indices 2 and 3 are not part of the stack content.
3. The stack is empty when length is 0, regardless of array content.

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Unbounded stack

An **unbounded stack** is a stack with no upper limit on length. An unbounded stack's length can increase indefinitely, so the stack's array allocation size must also be able to increase indefinitely.

PARTICIPATION ACTIVITY

5.18.2: Unbounded, array-based stack.



Animation content:

undefined

Animation captions:

1. The stack's initial allocation size is 1 and length is 0. So pushing 37 assigns array[0] with 37.
2. Pushing 88 occurs when allocationSize and length are both 1. So a new array is allocated, the existing entry is copied, and array[1] is assigned with 88.
3. Pushing 71 occurs when allocationSize and length are both 2. So a new array is allocated, the existing entries are copied, and array[2] is assigned with 71.

PARTICIPATION ACTIVITY

5.18.3: Unbounded stack.



Refer to the example above.

- 1) In the example above, when the array is reallocated, the size ____.



- increases by 1
- doubles
- decreases by 1

- 2) When does a push operation resize the stack's array?



- When `length == allocationSize`
- When `length == allocationSize - 1`
- Every push operation resizes the stack's array

- 3) In theory, an unbounded stack can grow in length indefinitely. In reality, the stack can ____.



- also grow in length indefinitely
- grow only until all distinct 32-bit integer values are pushed
- grow only until the resize operation fails to allocate memory

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Bounded stack

A **bounded stack** is a stack with a length that does not exceed a maximum value. The maximum is commonly the initial allocation size. Ex: A bounded stack with allocationSize = 100 cannot exceed a length of 100 items.

A bounded stack with a length equal to the maximum length is said to be **full**.

PARTICIPATION
ACTIVITY

5.18.4: Bounded, array-based stack.

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea
COLORADO CSPB2270Summer2023



Animation content:

undefined

Animation captions:

1. One implementation approach for a bounded stack is to allocate the array to the maximum length upon construction.
2. Three values push successfully.
3. The fourth push fails because length equals maxLength.
4. An alternate bounded stack implementation may distinguish allocation size and max length.
5. The first push assigns array[0] with 53. The second push must resize first, increasing allocationSize from 1 to 2.
6. Pushing 91 can now complete.
7. The next push must also resize. The $\min(\text{allocationSize} * 2, \text{maxLength}) = 3$ is the new allocation size.
8. The next push fails because length equals maxLength.

PARTICIPATION
ACTIVITY

5.18.5: Bounded stack - general implementation.



- 1) A bounded stack's maximum length and initial allocation size are always equal.

- True
- False

- 2) A bounded stack implementation may throw an exception if a push operation occurs when full.

- True
- False

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea
COLORADO CSPB2270Summer2023





- 3) Array-based stack implementations commonly reallocate when popping.

- True
- False

PARTICIPATION ACTIVITY

5.18.6: Bounded stack - two implementation approaches

Books 05/30/23 21:55 1692462

Taylor Larrechea

COLORADO CSPB2270Summer2023



Consider two bounded stack implementation approaches.

Implementation A: Allocation size is provided at construction time and is the stack's maximum length.

Implementation B: Maximum length is specified at construction time. Default allocation size is 1.

- 1) Which implementation(s) may need to reallocate the array to push an item?

- Implementation A only
- Implementation B only
- Both
- Neither



- 2) Which implementation(s) may allow a push when `length == maxLength`?

- Implementation A only
- Implementation B only
- Neither



- 3) Which implementation(s) guarantee that both push and pop execute in worst-case $O(1)$ time?

- Implementation A only
- Implementation B only
- Both
- Neither



©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADO CSPB2270Summer2023

Single implementation that allows bounded or unbounded

An array-based stack implementation can support both bounded and unbounded stack operations by using `maxLength` as follows:

- If `maxLength` is negative, the stack is unbounded.

- If `maxLength` is nonnegative, the stack is bounded.

The push operation does not allow a push when `length == maxLength`. The stack's length is always nonnegative, so:

- If `maxLength` is negative, the condition is never true. So all push operations are allowed, and therefore the stack is unbounded.
- If `maxLength` is nonnegative, the condition is true when the stack is full. So push operations are not allowed when full, and therefore the stack is bounded.

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

5.18.7: Stack push, resize, and pop operations.

**Animation content:**

undefined

Animation captions:

1. `ArrayStackPush` begins with a max length check. $0 \neq 3$, so the stack's length does not equal the maximum.
2. The stack's length, 0, does not equal allocation size, 1. So no resize is needed.
3. Array element 0 is assigned with 79, length is increased to 1, and true is returned.
4. When pushing 16, length and allocationSize are both 1, so `ArrayStackResize()` is called. The new allocation size is $1 * 2 = 2$.
5. 79 is copied, the stack's array and allocationSize are reassigned, and then execution returns to `ArrayStackPush` to complete the push.
6. Length and allocation size are both 2, so pushing 54 requires a resize. $\min(2 * 2, 3) = 3$ is the new allocation size.
7. The next push fails because length equals `maxLength`.
8. Popping 54 decreases the stack's length, but not allocation size. Popping does not resize the stack.

PARTICIPATION ACTIVITY

5.18.8: `ArrayStackInitialize()` function.



Assume the push, resize, and pop operations are implemented as in the animation above.

Consider an `ArrayStackInitialize()` function that initializes the stack before any operations occur.

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 1) If `maxLength` is assigned with -1, then `ArrayStackInitialize()` may assign `allocationSize` with ____.

-1



0 2

- 2) If `maxLength` is assigned with 64, then
`ArrayStackInitialize()` may assign
`allocationSize` with ____.

 any integer any nonnegative integer any positive integer

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY**5.18.9: Using ArrayStack functions for an unbounded stack.**

Assume a stack is initialized as follows:

```
integerStack->allocationSize = 1
integerStack->array = Allocate array of size 1
integerStack->length = 0
integerStack->maxLength = -1
```

Assume the push, resize, and pop functions are implemented as in the animation above and that the operations below execute in question order.

- 1) `ArrayStackPush(integerStack,`

21) ____.

- does not change the stack and returns false
- causes an out-of-bounds write in the stack's array
- successfully pushes 21 and returns true

- 2) `ArrayStackPush(integerStack,`

78) ____.

- does not change the stack and returns false
- causes an out-of-bounds write in the stack's array
- resizes, pushes 78, and returns true

- 3) Two more calls to `ArrayStackPush()` occur. `integerStack`'s allocationSize,

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

length, and maxLength are ____,
respectively.

- 2, 2, and -1
 - 4, 4, and -1
 - 4, 4, and 4
- 4) If each resize operation succeeds, the next 100 ArrayStackPush() operations succeed.

- True
- False

- 5) ArrayStackPop(), ArrayStackPush(), and ArrayStackResize() work for both bounded and unbounded stacks.

- True
- False

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

5.18.10: Array-based stack operation complexity.

- 1) For an unbounded stack, what operations take O(1) time in the worst case?

- ArrayStackPop() only
- ArrayStackPush() and ArrayStackPop()
- ArrayStackPush(), ArrayStackPop(), and ArrayStackResize()

- 2) For a bounded stack that does *not* resize, what operations take O(1) time in the worst case?

- ArrayStackPush() only
- ArrayStackPop() only
- ArrayStackPush() and ArrayStackPop()

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

5.19 Queue abstract data type (ADT)



This section has been set as optional by your instructor.

Queue abstract data type

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADO CSPB2270Summer2023

A **queue** is an ADT in which items are inserted at the end of the queue and removed from the front of the queue. The queue **enqueue** operation inserts an item at the end of the queue. The queue **dequeue** operation removes and returns the item at the front of the queue. Ex: After the operations "Enqueue 7", "Enqueue 14", and "Enqueue 9", "Dequeue" returns 7. A second "Dequeue" returns 14. A queue is referred to as a **first-in first-out** ADT. A queue can be implemented using a linked list or an array.

A queue ADT is similar to waiting in line at the grocery store. A person enters at the end of the line and exits at the front. British English actually uses the word "queue" in everyday vernacular where American English uses the word "line".

PARTICIPATION ACTIVITY

5.19.1: Queue ADT.



Animation content:

undefined

Animation captions:

1. A new queue named "wQueue" is created. Items are enqueued to the end of the queue.
2. Items are dequeued from the front of the queue.

PARTICIPATION ACTIVITY

5.19.2: Queue ADT.



- 1) Given numQueue: 5, 9, 1 (front is 5)

What are the queue contents after
the following enqueue operation?

Type the queue as: 1, 2, 3

Enqueue(numQueue, 4)

Check

Show answer

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADO CSPB2270Summer2023



2) Given numQueue: 11, 22 (the front is 11)

What are the queue contents after the following enqueue operations?

Type the queue as: 1, 2, 3

Enqueue(numQueue, 28)

Enqueue(numQueue, 72)

Check**Show answer**

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

3) Given numQueue: 49, 3, 8

What is returned by the following dequeue operation?

Dequeue(numQueue)

Check**Show answer**

4) Given numQueue: 4, 8, 7, 1, 3

What is returned by the second dequeue operation?

Dequeue(numQueue)

Dequeue(numQueue)

Check**Show answer**

5) Given numQueue: 15, 91, 11

What is the queue after the following dequeue operation? Type the queue as: 1, 2, 3

Dequeue(numQueue)

Check**Show answer**

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023





6) Given numQueue: 87, 21, 43

What are the queue's contents after the following operations? Type the queue as: 1, 2, 3

Dequeue(numQueue)

Enqueue(numQueue, 6)

Enqueue(numQueue, 50)

Dequeue(numQueue)

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSBP2270Summer2023

Check

Show answer

Common queue ADT operations

Table 5.19.1: Some common operations for a queue ADT.

Operation	Description	Example starting with queue: 43, 12, 77 (front is 43)
Enqueue(queue, x)	Inserts x at end of the queue	Enqueue(queue, 56). Queue: 43, 12, 77, 56
Dequeue(queue)	Returns and removes item at front of queue	Dequeue(queue) returns: 43. Queue: 12, 77
Peek(queue)	Returns but does not remove item at the front of the queue	Peek(queue) return 43. Queue: 43, 12, 77
IsEmpty(queue)	Returns true if queue has no items	IsEmpty(queue) returns false.
GetLength(queue)	Returns the number of items in the queue	GetLength(queue) returns 3.

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSBP2270Summer2023

Note: Dequeue and Peek operations should not be applied to an empty queue; the resulting behavior may be undefined.

PARTICIPATION ACTIVITY

5.19.3: Common queue ADT operations.





1) Given rosterQueue: 400, 313, 270, 514, 119, what does GetLength(rosterQueue) return?

- 400
- 5

2) Which operation determines if the queue contains no items?

- IsEmpty
- Peek

3) Given parkingQueue: 1, 8, 3, what are the queue contents after Peek(parkingQueue)?

- 1, 8, 3
- 8, 3

4) Given parkingQueue: 2, 9, 4, what are the contents of the queue after Dequeue(parkingQueue)?

- 9, 4
- 2, 9, 4

5) Given that parkingQueue has no items (i.e., is empty), what does GetLength(parkingQueue) return?

- 1
- 0
- Undefined



©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



CHALLENGE ACTIVITY

5.19.1: Queue ADT.



489394.3384924.qx3zqy7

Start

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Given numQueue: 96, 24

What are the queue's contents after the following operations?

- Enqueue(numQueue, 28)
- Enqueue(numQueue, 40)

Dequeue(numQueue)
~~Dequeue(numQueue)~~

After the given operations, what does GetLength(numQueue) return?

Ex: 8

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



5.20 Queues using linked lists



This section has been set as optional by your instructor.

A queue is often implemented using a linked list, with the list's head node representing the queue's front, and the list's tail node representing the queue's end. Enqueueing an item is performed by creating a new list node, assigning the node's data with the item, and appending the node to the list. Dequeueing is performed by assigning a local variable with the head node's data, removing the head node from the list, and returning the local variable.

PARTICIPATION ACTIVITY

5.20.1: Queue implemented using a linked list.



Animation content:

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

undefined

Animation captions:

1. Enqueueing an item puts the item in a list node and appends the node to the list.
2. A dequeue stores the head node's data in a local variable, removes the list's head node, and returns the local variable.

**PARTICIPATION
ACTIVITY**

5.20.2: Queue push and pop operations with a linked list.



Assume the queue is implemented using a linked list.

1) If the head pointer is null, the queue



- is empty
- is full
- has at least one item

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADO CSPB2270Summer2023

2) For the operation



QueueDequeue(queue), what is the second parameter passed to

ListRemoveAfter?

- The list's head node
- The list's tail node
- null

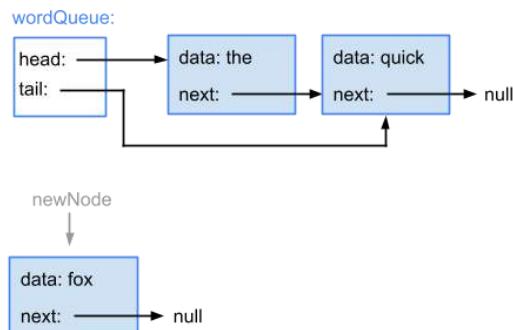
3) For the operation



QueueDequeue(queue), headData is assigned with the list ____ node's data.

- head
- tail

4) For QueueEnqueue(wordQueue, "fox"), which pointer is updated to point to the node?



©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADO CSPB2270Summer2023

- wordQueue's head pointer
- The head node's next pointer
- The tail node's next pointer

**CHALLENGE
ACTIVITY****5.20.1: Queues using linked lists.**

489394.3384924.qx3zqy7

Start

Given an empty queue numQueue, what does the list head pointer point to? If the pointer is null, enter null.

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Ex: 5 or null

What does the list tail pointer point to?

After the following operations:

QueueEnqueue(numQueue, 44)

QueueEnqueue(numQueue, 94)

QueueDequeue(numQueue)

What does the list head pointer point to?

What does the list tail pointer point to?

1**2****Check****Next**

5.21 Array-based queues

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Array-based queue storage

A queue can be implemented with an array. Three variables are needed in addition to the array:

- allocationSize: an integer for the array's allocated size.
- length: an integer for the number of items in the queue.
- frontIndex: an integer for the queue's front item index.

The queue's content starts at `array[frontIndex]` and continues forward through `length` items. If the array's end is reached before encountering all items, remaining items are stored starting at index 0.

PARTICIPATION ACTIVITY

5.21.1: Array-based queue storage.

**Animation content:**

undefined

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Animation captions:

1. Queue content starts at `frontIndex`. When `frontIndex` is 0 and `length` equals `allocationSize`, the queue's content matches the array's content: 78, 64, 16, 95.
2. If `frontIndex` is 2, then the queue's first two items are 16 and 95. Looping back to 0 gives the last two items, 78 and 64.
3. A `length` of 3 is one less than the allocation size, so 64 is not part of the queue.
4. The queue is empty when `length` is 0, regardless of array content.

PARTICIPATION ACTIVITY

5.21.2: Array-based queue storage.



A queue's array is [67, 19, 44, 38] and `allocationSize` is 4.

- 1) What is the queue's front item if `frontIndex` is 3 and `length` is 2?

- 67
- 44
- 38



- 2) What is the queue's back item if `frontIndex` is 0 and `length` is 3?

- 67
- 44
- 38



- 3) What is the queue's content if `length` is 4 and `frontIndex` is 1?

- (front) 67, 19, 44, 38 (back)
- (front) 19, 44, 38 (back)
- (front) 19, 44, 38, 67 (back)

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Bounded vs. unbounded queue

A **bounded queue** is a queue with a length that does not exceed a specified maximum value. An additional variable, `maxLength`, is needed. `maxLength` is commonly assigned at construction time and does not change for the queue's lifetime. A bounded queue with a length equal to the maximum length is said to be **full**.

An **unbounded queue** is a queue with a length that can grow indefinitely.

zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADO CSPB2270Summer2023

PARTICIPATION
ACTIVITY

5.21.3: Bounded vs. unbounded queue.



Animation content:

undefined

Animation captions:

1. Three queues are created: the first unbounded, the second bounded with a maximum length of 2, and the third bounded with a maximum length of 4.
2. Enqueueing 51 then 88 yields the same results for each queue.
3. Enqueueing 73 fails for the full bounded queue, because the length and maximum length are both 2. Enqueueing 73 succeeds for the other two queues.
4. Similarly, enqueueing 47 succeeds for the first and third queues and fails for the second.
5. Both bounded queues now have a length equal to the maximum length. So enqueueing 24 succeeds only for the unbounded queue.

PARTICIPATION
ACTIVITY

5.21.4: Bounded vs. unbounded queue.



1) ____ can be full.



- Only a bounded queue
- Only an unbounded queue
- Both bounded and unbounded queues

2) A bounded queue implementation may throw an exception if an enqueue operation occurs when full.

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADO CSPB2270Summer2023

- True
- False

3) The `maxLength` variable is needed



- only for the bounded queue implementation
- only for the unbounded queue implementation
- for both the bounded and unbounded queue implementations

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Flexible implementation and resize operation

An array-based queue implementation can support both bounded and unbounded queue operations by using `maxLength` as follows:

- If `maxLength` is negative, the queue is unbounded.
- If `maxLength` is nonnegative, the queue is bounded.

The resize operation allocates an array of larger size, commonly double the existing allocation size. If `maxLength` is nonnegative, the minimum of `maxLength` and double the current allocation size is used. Existing array entries are copied in such a way that the front index is reset to 0.

PARTICIPATION ACTIVITY

5.21.5: Array-based queue resize operation.



Animation content:

undefined

Animation captions:

1. `maxLength = 4`, so `queue1` is bounded. `length` and `allocationSize` are both 3, so `queue1` is full.
2. `frontIndex` is 1, so `queue1`'s content from front to back is: 16, 53, 97.
3. Resizing `queue1` begins with computing `newSize`. Double the current allocation size is 6. But `maxLength` is 4, which is nonnegative and less than 6. So 4 is used instead.
4. The new array is allocated. The first element is at index 1 in the existing array and is copied to index 0 in the new array.
5. Remaining elements are copied. Then array, `allocationSize`, and `frontIndex` are reassigned.
6. `queue2`'s `maxLength` is -1 and `maxLength` is never reassigned. So when resizing, the condition `queue->maxLength >= 0` is always false.
7. So `queue2` is unbounded, and each resize doubles the allocation size.

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

5.21.6: Flexible implementation and resize operation.



Consider `queueA` with variables:

```
allocationSize: 4
array: [73, 91, 87, 62]
frontIndex: 2
length: 4
maxLength: -1
```

1) queueA is ____.

- bounded and full
- bounded and not full
- unbounded



©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

2) After executing

`ArrayQueueResize(queueA),`
queueA's allocation size is ____.

- 1
- 4
- 8



3) After executing

`ArrayQueueResize(queueA),`
queueA.queue_list's first four
elements are ____.

- 73, 91, 87, 62
- 87, 62, 73, 91



4) Can one of queueA's initial variables be
changed such that executing

`ArrayQueueResize(queueA)`
makes queueA's allocation size 6?

- Yes, if `maxLength` were initially 6
- Yes, if `allocationSize` were
initially 6
- No



Enqueue and dequeue operations

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

An enqueue operation:

1. Compares `length` and `maxLength`. If equal, the queue is full and so no change occurs and false is returned.
2. Compares `length` and `allocationSize`. If equal, a resize operation occurs.
3. Computes the enqueued item's index as `(frontIndex + length) % allocationSize` and assigns to the array at that index. Ex: Enqueueing 42 into a queue with `frontIndex` 2, `length` 3, and

allocationSize 8 assigns the queue array at index $(2 + 3) \% 8 = 5$ with 42.

4. Increments `length` and then returns true.

A dequeue operation:

1. Makes a copy of the array item at `frontIndex`.
2. Decrements `length`.
3. Increments `frontIndex`, resetting to 0 if the incremented value equals the allocation size.
4. Returns the array item from step 1.

©zyBooks 05/30/23 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

5.21.7: Array-based queue enqueue and dequeue operations.



Animation content:

undefined

Animation captions:

1. `numQueue` is an empty queue with `allocationSize = 1`. Enqueuing 42 starts by comparing `length` against `maxLength`, then allocation size.
2. `array[0]` is assigned with 42, `length` is incremented, and true is returned.
3. When enqueueing 89, a resize occurs first. Then `array[1]` is assigned with 89.
4. Enqueueing 71 resizes again, increasing `allocationSize` to 4. Then 64 is enqueued without resizing.
5. `frontIndex = 0`, so the dequeue operation begins by assigning `toReturn` with `array[0]`. Length is decremented to 3, `frontIndex` is reassigned with 1, and 42 is returned.
6. 89 is dequeued similarly.
7. 91 is enqueued at index 0.

PARTICIPATION ACTIVITY

5.21.8: Enqueue and dequeue operations.



1) ____ may resize the queue.



- Only the enqueue operation
- Only the dequeue operation
- Both the enqueue and dequeue operations

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

2) `(frontIndex + length) % allocationSize` yields the index



- of the item to remove during a dequeue operation

- at which to insert a new item during an enqueue operation
 - of the queue's back item
- 3) ArrayQueueEnqueue() returns false if _____.

- the item is successfully enqueued
- the resize operation fails
- the queue is full

- 4) An alternate implementation could store the queue's back item index instead of the length.

- True
- False

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Worst-case time complexity

Using the implementation from above, worst-case time complexities are the same whether the queue is bounded or unbounded: $O(N)$ for enqueue and $O(1)$ for dequeue.

An alternate implementation of the bounded queue, not presented in this section, uses `maxLength` as the array's initial allocation size. Such an implementation never needs to resize and so the enqueue operation's worst-case time is $O(1)$.

5.22 Deque abstract data type (ADT)



This section has been set as optional by your instructor.

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Deque abstract data type

A **deque** (pronounced "deck" and short for double-ended queue) is an ADT in which items can be inserted and removed at both the front and back. The deque push-front operation inserts an item at the front of the deque, and the push-back operation inserts at the back of the deque. The pop-front operation removes and returns the item at the front of the deque, and the pop-back operation removes and returns the item at the back of the deque. Ex: After the operations "push-back 7", "push-front 14", "push-front 9",

and "push-back 5", "pop-back" returns 5. A subsequent "pop-front" returns 9. A deque can be implemented using a linked list or an array.

PARTICIPATION ACTIVITY

5.22.1: Deque ADT.

**Animation captions:**

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

1. The "push-front 34" operation followed by "push-front 51" produces a deque with contents 51, 34.
2. The "push-back 19" operation pushes 19 to the back of the deque, yielding 51, 34, 19. "Pop-front" then removes and returns 51.
3. Items can also be removed from the back of the deque. The "pop-back" operation removes and returns 19.

PARTICIPATION ACTIVITY

5.22.2: Deque ADT.



Determine the deque contents after the following operations.

If unable to drag and drop, refresh the page.

**push-front 97,
push-back 71,
pop-front,
push-front 45,
push-back 68**

**push-back 45,
push-back 71,
push-front 97,
push-front 68,
pop-back**

**push-front 71,
push-front 68,
push-front 97,
pop-back,
push-front 45**

45, 97, 68

45, 71, 68

68, 97, 45

Reset

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Common deque ADT operations

In addition to pushing or popping at the front or back, a deque typically supports peeking at the front and back of the deque and determining the length. A **peek** operation returns an item in the deque without removing the item.

Table 5.22.1: Common deque ADT operations.

Operation	Description	Example starting with deque: 59, 63, 19 (front is 59)
PushFront(deque, x)	Inserts x at the front of the deque	PushFront(deque, 41). Deque: 41, 59, 63, 19 ©zyBooks 05/30/23 21:55 1692462 Taylor Larrechea COLORADOCSPB2270Summer2023
PushBack(deque, x)	Inserts x at the back of the deque	PushBack(deque, 41). Deque: 59, 63, 19, 41
PopFront(deque)	Returns and removes item at front of deque	PopFront(deque) returns 59. Deque: 63, 19
PopBack(deque)	Returns and removes item at back of deque	PopBack(deque) returns 19. Deque: 59, 63
PeekFront(deque)	Returns but does not remove the item at the front of deque	PeekFront(deque) returns 59. Deque is still: 59, 63, 19
PeekBack(deque)	Returns but does not remove the item at the back of deque	PeekBack(deque) returns 19. Deque is still: 59, 63, 19
IsEmpty(deque)	Returns true if the deque is empty	IsEmpty(deque) returns false.
GetLength(deque)	Returns the number of items in the deque	GetLength(deque) returns 3.

PARTICIPATION ACTIVITY

5.22.3: Common deque ADT operations.



- 1) Given rosterDeque: 351, 814, 216, 636, 484, 102, what does GetLength(rosterDeque) return?

- 351
- 102
- 6

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



- 2) Which operation determines if the deque contains no items?

- IsEmpty
- PeekFront





- 3) Given jobsDeque: 4, 7, 5, what are the deque contents after PeekBack(jobsDeque)?

- 4, 7, 5
- 4, 7

- 4) Given jobsDeque: 3, 6, 1, 7, what are the contents of the deque after PopFront(jobsDeque)?

- 6, 1, 7
- 3, 6, 1, 7

- 5) Given that jobsDeque is empty, what does GetLength(jobsDeque) return?

- 1
- 0
- Undefined

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



CHALLENGE ACTIVITY

5.22.1: Deque ADT.



489394.3384924.qx3zqy7

Start

Given an empty deque numDeque, what are the deque's contents after the following operation

- PushFront(numDeque, 49)
- PushBack(numDeque, 70)
- PushBack(numDeque, 58)
- PushFront(numDeque, 65)

Ex: 1, 2, 3 (commas between values)

After the above operations, what does PeekFront(numDeque) return?

Ex: 5

After the above operations, what does PeekBack(numDeque) return?

©zyBooks 05/30/23 21:55 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

After the above operations, what does GetLength(numDeque) return?

1

2

3

Check

Next

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

©zyBooks 05/30/23 21:55 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

6.1 Recursion: Introduction

An **algorithm** is a sequence of steps for solving a problem. For example, an algorithm for making lemonade is:

Figure 6.1.1: Algorithms are like recipes.



Make lemonade:

- Add sugar to pitcher
- Add lemon juice
- Add water
- Stir

©zyBooks 06/06/23 23:52 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Some problems can be solved using a recursive algorithm. A **recursive algorithm** is an algorithm that breaks the problem into smaller subproblems and applies the same algorithm to solve the smaller subproblems.

Figure 6.1.2: Mowing the lawn can be broken down into a recursive process.



- Mow the lawn
 - Mow the frontyard
 - Mow the left front
 - Mow the right front
 - Mow the backyard
 - Mow the left back
 - Mow the right back

The mowing algorithm consists of applying the mowing algorithm on smaller pieces of the yard and thus is a recursive algorithm.

At some point, a recursive algorithm must describe how to actually do something, known as the **base case**. The mowing algorithm could thus be written as:

- Mow the lawn
 - If lawn is less than 100 square meters
 - Push the lawnmower left-to-right in adjacent rows
 - Else
 - Mow one half of the lawn
 - Mow the other half of the lawn

©zyBooks 06/06/23 23:52 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

PARTICIPATION
ACTIVITY

6.1.1: Recursion.



Which are recursive definitions/algorithms?



1) Helping N people:

If N is 1, help that person.
 Else, help the first N/2 people, then help the second N/2 people.

- True
- False

2) Driving to the store:

©zyBooks 06/23 23:52 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

Go 1 mile.
 Turn left on Main Street.
 Go 1/2 mile.

- True
- False

3) Sorting envelopes by zipcode:



If N is 1, done.
 Else, find the middle zipcode. Put all zipcodes less than the middle zipcode on the left, all greater ones on the right.
 Then sort the left, then sort the right.

- True
- False

6.2 Recursive functions

A function may call other functions, including calling itself. A function that calls itself is a **recursive function**.

PARTICIPATION
ACTIVITY

6.2.1: A recursive function example.



Animation captions:

1. The first call to CountDown() function comes from main. Each call to CountDown() effectively creates a new "copy" of the executing function, as shown on the right.
2. Then, the CountDown() function calls itself. CountDown(1) similarly creates a new "copy" of the executing function.
3. CountDown() function calls itself once more.
4. That last instance does not call CountDown() again, but instead returns. As each instance returns, that copy is deleted.

06/06/23 23:52 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

Each call to CountDown() effectively creates a new "copy" of the executing function, as shown on the right. Returning deletes that copy.

The example is for demonstrating recursion; counting down is otherwise better implemented with a loop.

Recursion may be direct, such as f() itself calling f(), or indirect, such as f() calling g() and g() calling f().

PARTICIPATION ACTIVITY

6.2.2: Thinking about recursion.



Refer to the above CountDown example for the following.

- 1) How many times is CountDown()
called if main() calls CountDown(5)?

**Check****Show answer**

©zyBooks 06/06/23 23:52 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 2) How many times is CountDown()
called if main() calls CountDown(0)?

**Check****Show answer**

- 3) Is there a difference in how we
define the parameters of a recursive
versus non-recursive function?



Answer yes or no.

Check**Show answer****CHALLENGE ACTIVITY**

6.2.1: Calling a recursive function.



Write a statement that calls the recursive function BackwardsAlphabet() with parameter startingLetter.

[Learn how our autograder works](#)

489394.3384924.qx3zqy7

```

1 #include <iostream>
2 using namespace std;
3
4 void BackwardsAlphabet(char currLetter){
5     if (currLetter == 'a') {
6         cout << currLetter << endl;
7     }
8     else{
9         cout << currLetter << " ";
10        BackwardsAlphabet(currLetter - 1);
11    }
12 }
13
14 int main(){
15     char startingLetter;

```

©zyBooks 06/06/23 23:52 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Run

View your last submission ▾

6.3 Recursive algorithm: Search

Recursive search (general)

Consider a guessing game program where a friend thinks of a number from 0 to 100 and you try to guess the number, with the friend telling you to guess higher or lower until you guess correctly. What algorithm would you use to minimize the number of guesses?

A first try might implement an algorithm that simply guesses in increments of 1:

- Is it 0? Higher
- Is it 1? Higher
- Is it 2? Higher

This algorithm requires too many guesses (50 on average). A second try might implement an algorithm that guesses by 10s and then by 1s:

- Is it 10? Higher
- Is it 20? Higher
- Is it 30? Lower
- Is it 21? Higher
- Is it 22? Higher
- Is it 23? Higher

This algorithm does better but still requires about 10 guesses on average: 5 to find the correct tens digit and 5 to guess the correct ones digit. An even better algorithm uses a binary search. A **binary search** algorithm begins at the midpoint of the range and halves the range after each guess. For example:

- Is it 50 (the middle of 0-100)? Lower
- Is it 25 (the middle of 0-50)? Higher
- Is it 38 (the middle of 26-50)? Lower
- Is it 32 (the middle of 26-38)?

After each guess, the binary search algorithm is applied again, but on a smaller range, i.e., the algorithm is recursive.

PARTICIPATION ACTIVITY

6.3.1: Binary search: A well-known recursive algorithm.



Animation content:

undefined

Animation captions:

©zyBooks 06/06/23 23:52 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

1. A friend thinks of a number from 0 to 100 and you try to guess the number, with the friend telling you to guess higher or lower until you guess correctly.
2. Using a binary search algorithm, you begin at the midpoint of the lower range. ($\text{highVal} + \text{lowVal} / 2 = (100 + 0) / 2$, or 50).
3. The number is lower. The algorithm divides the range in half, then chooses the midpoint of that range.
4. After each guess, the binary search algorithm is applied, halving the range and guessing the midpoint or the corresponding range.

5. A recursive function is a natural match for the recursive binary search algorithm. A function GuessNumber(lowVal, highVal) has parameters that indicate the low and high sides of the guessing range.

Recursive search function

A recursive function is a natural match for the recursive binary search algorithm. A function GuessNumber(lowVal, highVal) has parameters that indicate the low and high sides of the guessing range. The function guesses at the midpoint of the range. If the user says lower, the function calls GuessNumber(lowVal, midVal). If the user says higher, the function calls GuessNumber(midVal + 1, highVal)

The recursive function has an if-else statement. The if branch ends the recursion, known as the **base case**. The else branch has recursive calls. Such an if-else pattern is common in recursive functions.

Figure 6.3.1: A recursive function carrying out a binary search algorithm.

```
#include <iostream>
using namespace std;

void GuessNumber(int lowVal, int highVal) {
    int midVal; // Midpoint of low and high value
    char userAnswer; // User response

    midVal = (highVal + lowVal) / 2;

    // Prompt user for input
    cout << "Is it " << midVal << "? (l/h/y): ";
    cin >> userAnswer;

    if( (userAnswer != 'l') && (userAnswer != 'h') ) { // Base case: found
        number
        cout << "Thank you!" << endl;
    }
    else { // Recursive case:
        split into lower OR upper half
        if (userAnswer == 'l') { // Guess in lower half
            GuessNumber(lowVal, midVal); // Recursive call
        }
        else { // Guess in upper half
            GuessNumber(midVal + 1, highVal); // Recursive call
        }
    }
}

int main() {
    // Print game objective, user input commands
    cout << "Choose a number from 0 to 100." << endl;
    cout << "Answer with:" << endl;
    cout << "    l (your num is lower)" << endl;
    cout << "    h (your num is higher)" << endl;
    cout << "    any other key (guess is right)." << endl;

    // Call recursive function to guess number
    GuessNumber(0, 100);

    return 0;
}
```

©zyBooks 06/06/23 23:52 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

```
Choose a number from 0 to 100.
Answer with:
    l (your num is lower)
    h (your num is higher)
    any other key (guess is right).
Is it 50? (l/h/y): l
Is it 25? (l/h/y): h
Is it 38? (l/h/y): l
Is it 32? (l/h/y): y
Thank you!
```

Calculating the middle value

Because `midVal` has already been checked, it need not be part of the new window, so `midVal + 1` rather than `midVal` is used for the window's new low side, or `midVal - 1` for the window's new high side. But the `midVal - 1` can have the drawback of a non-intuitive base case (i.e., `midVal < lowVal`, because if the current window is say 4..5, `midVal` is 4, so the new window would be 4..4-1, or 4..3). `rangeSize == 1` is likely more intuitive, and thus the algorithm uses `midVal` rather than `midVal - 1`. However, the algorithm uses `midVal + 1` when searching higher, due to integer rounding. In particular, for window 99..100, `midVal` is 99 ($(99 + 100) / 2 = 99.5$, rounded to 99 due to truncation of the fraction in integer division). So the next window would again be 99..100, and the algorithm would repeat

©zyBooks 06/06/23 23:52 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

with this window forever. $\text{midVal} + 1$ prevents the problem, and doesn't miss any numbers because midVal was checked and thus need not be part of the window.

**PARTICIPATION
ACTIVITY**
6.3.2: Binary search tree tool.


The following program guesses the hidden number known by the user. Assume the hidden number is 63.

Books 06/06/23 23:52 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

```
#include <iostream>
using namespace std;

void Find(int low, int high) {
    int mid; // Midpoint of low..high
    char answer;

    mid = (high + low) / 2;

    cout << "Is it " << mid << "? (l/h/y): ";
    cin >> answer;

    if((answer != 'l') &&
       (answer != 'h')) { // Base case:
        cout << "Thank you!" << endl; // Found number!
    }
    else { // Recursive case: Guess in
           // lower or upper half of range
        if (answer == 'l') { // Guess in lower half
            Find(low, mid); // Recursive call
        }
        else { // Guess in upper half
            Find(mid + 1, high); // Recursive call
        }
    }

    return;
}

int main() {
    cout << "Choose a number from 0 to 100." << endl;
    cout << "Answer with: " << endl;
    cout << "    l (your num is lower)" << endl;
    cout << "    h (your num is higher)" << endl;
    cout << "    any other key (guess is right)." << endl;

    Find(0, 100);

    return 0;
}
```

main()

```
int main() {
    cout << "Choose a number from 0 to 100." << endl;
    cout << "Answer with: " << endl;
    cout << "    l (your num is lower)" << endl;
    cout << "    h (your num is higher)" << endl;
    cout << "    any other key (guess is right)." << endl;

    Find(0, 100);

    return 0;
}
```

©zyBooks 06/06/23 23:52 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

©zyBooks 06/06/23 23:52 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Recursively searching a sorted list

Search is commonly performed to quickly find an item in a sorted list stored in an array or vector. Consider a list of attendees at a conference, whose names have been stored in alphabetical order in an array or vector. The following quickly determines whether a particular person is in attendance.

FindMatch() restricts its search to elements within the range lowVal to highVal. main() initially passes a range of the entire list: 0 to (list size - 1). FindMatch() compares to the middle element, returning that element's position if matching. If not matching, FindMatch() checks if the window's size is just one element, returning -1 in that case to indicate the item was not found. If neither of those two base cases are satisfied, then FindMatch() recursively searches either the lower or upper half of the range as appropriate.

Figure 6.3.2: Recursively searching a sorted list.

©zyBooks 06/06/23 23:52 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

©zyBooks 06/06/23 23:52 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

©zyBooks 06/06/23 23:52 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

```

#include <iostream>
#include <string>
#include <vector>

using namespace std;

/* Finds index of string in vector of strings, else -1.
   Searches only with index range low to high
   Note: Upper/lower case characters matter
*/

int FindMatch(vector<string> stringsList, string itemMatch, int lowVal, int highVal) {
    int midVal;           // Midpoint of low and high values
    int itemPos;          // Position where item found, -1 if not found
    int rangeSize;        // Remaining range of values to search for match

    rangeSize = (highVal - lowVal) + 1;
    midVal = (highVal + lowVal) / 2;

    if (itemMatch == stringsList.at(midVal)) {      // Base case 1: item found
at midVal position
        itemPos = midVal;
    }
    else if (rangeSize == 1) {                         // Base case 2: match not
found
        itemPos = -1;
    }
    else {                                              // Recursive case: search
lower or upper half
        if (itemMatch < stringsList.at(midVal)) { // Search lower half,
recursive call
            itemPos = FindMatch(stringsList, itemMatch, lowVal, midVal);
        }
        else {                                         // Search upper half,
recursive call
            itemPos = FindMatch(stringsList, itemMatch, midVal + 1, highVal);
        }
    }
}

return itemPos;
}

int main() {
    vector<string> attendeesList(0); // List of attendees
    string attendeeName;           // Name of attendee to match
    int matchPos;                 // Matched position in attendee list

    // Omitting part of program that adds attendees
    // Instead, we insert some sample attendees in sorted order
    attendeesList.push_back("Adams, Mary");
    attendeesList.push_back("Carver, Michael");
    attendeesList.push_back("Domer, Hugo");
    attendeesList.push_back("Fredericks, Carlos");
    attendeesList.push_back("Li, Jie");

    // Prompt user to enter a name to find
    cout << "Enter person's name: Last, First: ";
    getline(cin, attendeeName); // Use getline to allow space in name

    // Call function to match name, output results
    matchPos = FindMatch(attendeesList, attendeeName, 0, attendeesList.size()
- 1);
    if (matchPos >= 0) {
        cout << "Found at position " << matchPos << "." << endl;
    }
    else {
        cout << "Not found. " << endl;
    }

    return 0;
}

```

@zyBooks 06/06/23 23:52 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

```
Enter person's name: Last, First: Meeks, Stan
Not found.

...
Enter person's name: Last, First: Adams, Mary
Found at position 0.

...
Enter person's name: Last, First: Li, Jie
Found at position 4.
```

©zyBooks 06/06/23 23:52 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

6.3.3: Recursive search algorithm.



Consider the above FindMatch() function for finding an item in a sorted list.

- 1) If a sorted list has elements 0 to 50 and the item being searched for is at element 6, how many times will FindMatch() be called?

Check
[Show answer](#)


- 2) If an alphabetically ascending list has elements 0 to 50, and the item at element 0 is "Bananas", how many calls to FindMatch() will be made during the failed search for "Apples"?

Check
[Show answer](#)

PARTICIPATION ACTIVITY

6.3.4: Recursive calls.



A list has 5 elements numbered 0 to 4, with these letter values: 0: A, 1: B, 2: D, 3: E, 4: F.

- 1) To search for item C, the first call is FindMatch(0, 4). What is the second call to FindMatch()?

- FindMatch(0, 0)
- FindMatch(0, 2)
- FindMatch(3, 4)

©zyBooks 06/06/23 23:52 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023



- 2) In searching for item C, FindMatch(0, 2) is called. What happens next?

- Base case 1: item found at midVal.



- Base case 2: rangeSize == 1, so no match.
- Recursive call: FindMatch(2, 2)

CHALLENGE ACTIVITY

6.3.1: Enter the output of binary search.



489394.3384924.qx3zqy7

Start

©zyBooks 06/06/23 23:52 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Type the program's output

```
#include <iostream>
using namespace std;

void FindNumber(int number, int lowVal, int highVal) {
    int midVal;

    midVal = (highVal + lowVal) / 2;
    cout << number;
    cout << " ";
    cout << midVal;

    if (number == midVal) {
        cout << " q" << endl;
    }
    else {
        if (number < midVal) {
            cout << " r" << endl;
            FindNumber(number, lowVal, midVal);
        }
        else {
            cout << " s" << endl;
            FindNumber(number, midVal + 1, highVal);
        }
    }
}

int main() {
    int number;

    cin >> number;
    FindNumber(number, 0, 10);

    return 0;
}
```

Input

1

Output



1

2

Check**Next****CHALLENGE ACTIVITY**

6.3.2: Recursive algorithm: Search.



489394.3384924.qx3zqy7

Start

©zyBooks 06/06/23 23:52 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Integer numData is read from input. Then, numData alphabetically sorted characters are read from input and appended to a vector. Complete the FindData() function, which outputs rangeSize, midIndex, and the element at index midIndex.

- Assign rangeSize with the total number of vector elements from lowerIndex to upperIndex (both inclusive).
- Assign midIndex with the result of dividing the sum of lowerIndex and upperIndex by 2.

Ex: If the input is:

```
5  
f h k m y
```

then the output is:

```
Number of elements in the range: 5  
Middle index: 2  
Element at middle index: k
```

```
1 #include <iostream>  
2 #include <vector>  
3 using namespace std;  
4  
5 void FindData(vector<char> lettersVector, int lowerIndex, int upperIndex) {  
6     int midIndex;  
7     int rangeSize;  
8  
9     /* Your code goes here */  
10  
11    cout << "Number of elements in the range: " << rangeSize << endl;  
12    cout << "Middle index: " << midIndex << endl;  
13    cout << "Element at middle index: " << lettersVector.at(midIndex) << endl;  
14 }  
15
```

©zyBooks 06/06/23 23:52 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

1

2

Check

Next level

Exploring further:

- [Binary search](#) from GeeksforGeeks.org

6.4 Adding output statements for debugging

Recursive functions can be particularly challenging to debug. Adding output statements can be helpful. Furthermore, an additional trick is to indent the print statements to show the current depth of recursion. The following program adds a parameter indent to a `FindMatch()` function that searches a sorted list for an item. All of `FindMatch()`'s print statements start with `cout << indentAmt <<`. Indent is typically some number of spaces. `main()` sets indent to three spaces. Each recursive call adds three more spaces. Note how the output now clearly shows the recursion depth.

©zyBooks 06/06/23 23:52 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Figure 6.4.1: Output statements can help debug recursive functions, especially if indented based on recursion depth.

©zyBooks 06/06/23 23:52 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

©zyBooks 06/06/23 23:52 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

```

#include <iostream>
#include <string>
#include <vector>

using namespace std;

/* Finds index of string in vector of strings, else -1.
   Searches only with index range low to high
   Note: Upper/lower case characters matter
*/

int FindMatch(vector<string> stringsList, string itemMatch,
              int lowVal, int highVal, string indentAmt) { // indentAmt used
for print debug

    int midVal;           // Midpoint of low and high values
    int itemPos;          // Position where item found, -1 if not found
    int rangeSize;        // Remaining range of values to search for match

    cout << indentAmt << "Find() range " << lowVal << " " << highVal << endl;

    rangeSize = (highVal - lowVal) + 1;
    midVal = (highVal + lowVal) / 2;

    if (itemMatch == stringsList.at(midVal)) {    // Base case 1: item found
at midVal position
        cout << indentAmt << "Found person." << endl;
        itemPos = midVal;
    }
    else if (rangeSize == 1) {                      // Base case 2: match not
found
        cout << indentAmt << "Person not found." << endl;
        itemPos = -1;
    }
    else {                                         // Recursive case: Search
lower or upper half
        if (itemMatch < stringsList.at(midVal)) { // Search lower half,
recursive call
            cout << indentAmt << "Searching lower half." << endl;
            itemPos = FindMatch(stringsList, itemMatch, lowVal, midVal,
indentAmt + " ");
        }
        else {                                     // Search upper half,
recursive call
            cout << indentAmt << "Searching upper half." << endl;
            itemPos = FindMatch(stringsList, itemMatch, midVal + 1, highVal,
indentAmt + " ");
        }
    }

    cout << indentAmt << "Returning pos = " << itemPos << "." << endl;
    return itemPos;
}

int main() {
    vector<string> attendeesList(0); // List of attendees
    string attendeeName;           // Name of attendee to match
    int matchPos;                 // Matched position in attendee list

    // Omitting part of program that adds attendees
    // Instead, we insert some sample attendees in sorted order
    attendeesList.push_back("Adams, Mary");
    attendeesList.push_back("Carver, Michael");
    attendeesList.push_back("Domer, Hugo");
    attendeesList.push_back("Fredericks, Carlos");
    attendeesList.push_back("Li, Jie");

    // Prompt user to enter a name to find
    cout << "Enter person's name: Last, First: ";
    getline(cin, attendeeName); // Use getline to allow space in name

    // Call function to match name, output results
    matchPos = FindMatch(attendeesList, attendeeName, 0);
}

```

©zyBooks 06/06/23 23:52 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

©zyBooks 06/06/23 23:52 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

```

Enter person's name: Last, First: Meeks, Stan
Find() range 0 4
Searching upper half.
Find() range 3 4
Searching upper half.
Find() range 4 4
Person not found.
Returning pos = -1.
Returning pos = -1.
Returning pos = -1.
Not found.

...
Enter person's name: Last, First: Adams, Mary
Find() range 0 4
Searching lower half.
Find() range 0 2
Searching lower half.
Find() range 0 1
Found person.
Returning pos = 0.
Returning pos = 0.
Returning pos = 0.
Found at position 0.

```

©zyBooks 06/06/23 23:52 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

Some programmers like to leave the output statements in the code, commenting them out with "://" when not in use. The statements actually serve as a form of comment as well.

More advanced techniques for handling debug output exist too, such as **conditional compilation** (beyond this section's scope).

PARTICIPATION ACTIVITY

6.4.1: Recursive debug statements.



Refer to the above code using indented output statements.

- 1) The above debug approach requires an extra parameter be passed to indicate the amount of indentation.



- True
- False

- 2) Each recursive call should add a few spaces to the indent parameter.



- True
- False

- 3) The function should remove a few spaces from the indent parameter before returning.



- True
- False

©zyBooks 06/06/23 23:52 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

zyDE 6.4.1: Output statements in a recursive function.

- Run the recursive program, and observe the output statements for debugging, and see if the person is correctly not found.

- Introduce an error by changing `itemPos = -1` to `itemPos = 0` in the range size base case.
- Run the program, notice how the indented print statements help isolate the error of person incorrectly being found.

[Load default template...](#)
[Run](#)

```

1 #include <iostream>
2 #include <string>
3 #include <vector>
4
5 using namespace std;
6
7 /* Finds index of string in vector
8    Searches only with index range l
9    Note: Upper/lower case character
10   */
11
12
13 int FindMatch(vector<string> string
14     .... int lowVal, int highV
15     ....

```

©zyBooks 06/06/23 23:52 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

6.5 Creating a recursive function

Creating a recursive function can be accomplished in two steps.

- **Write the base case** -- Every recursive function must have a case that returns a value without performing a recursive call. That case is called the **base case**. A programmer may write that part of the function first, and then test. There may be multiple base cases.
- **Write the recursive case** -- The programmer then adds the recursive case to the function.

The following illustrates a simple function that computes the factorial of N (i.e. $N!$). The base case is $N = 1$ or $1!$ which evaluates to 1. The base case is written as `if (N <= 1) { fact = 1; }`. The recursive case is used for $N > 1$, and written as `else { fact = N * NFact(N - 1); }`.

PARTICIPATION
ACTIVITY

6.5.1: Writing a recursive function for factorial: First write the base case, then add the recursive case.



Animation captions:

1. The base case, which returns a value without performing a recursive call, is written and tested first. If N is less than or equal to 1, then the `NFact()` function returns 1.
2. Next the recursive case, which calls itself, is written and tested. If N is greater than 1, then the `NFact()` function returns $N * NFact(N - 1)$.

©zyBooks 06/06/23 23:52 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

A common error is to not cover all possible base cases in a recursive function. Another common error is to write a recursive function that doesn't always reach a base case. Both errors may lead to infinite recursion, causing the program to fail.

Typically, programmers will use two functions for recursion. An "outer" function is intended to be called from other parts of the program, like the function `int CalcFactorial(int inVal)`. An "inner" function is intended only to be called from that outer function, for example a function `int CalcFactorialHelper(int inVal)`. The outer function may check for a valid input value, e.g., ensuring `inVal` is not negative, and then calling the inner function. Commonly, the inner function has

parameters that are mainly of use as part of the recursion, and need not be part of the outer function, thus keeping the outer function more intuitive.

**PARTICIPATION
ACTIVITY**

6.5.2: Creating recursion.



- 1) Recursive functions can be accomplished in one step, namely repeated calls to itself.

 True

 False

- 2) A recursive function with parameter N counts up from any negative number to 0. An appropriate base case would be $N == 0$.

 True

 False

- 3) A recursive function can have two base cases, such as $N == 0$ returning 0, and $N == 1$ returning 1.

 True

 False

©zyBooks 06/06/23 23:52 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Before writing a recursive function, a programmer should determine:

1. Does the problem naturally have a recursive solution?
2. Is a recursive solution better than a non-recursive solution?

For example, computing $N!$ (N factorial) does have a natural recursive solution, but a recursive solution is not better than a non-recursive solution. The figure below illustrates how the factorial computation can be implemented as a loop. Conversely, binary search has a natural recursive solution, and that solution may be easier to understand than a non-recursive solution.

 Figure 6.5.1: Non-recursive solution to compute $N!$

```
for (i = inputNum; i > 1; --i)
{
    facResult = facResult * i;
}
```

**PARTICIPATION
ACTIVITY**

6.5.3: When recursion is appropriate.

©zyBooks 06/06/23 23:52 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

- 1) N factorial ($N!$) is commonly implemented as a recursive function due to being easier to understand and executing faster than a loop implementation.

 True

False

zyDE 6.5.1: Output statements in a recursive function.

Implement a recursive function to determine if a number is prime. Skeletal code is provided for the `IsPrime` function.

```
1
2 #include <iostream>
3 using namespace std;
4
5 // Returns false if value is not prime
6 bool IsPrime(int testVal, int divVal)
7 {
8     // Base case 1: 0 and 1 are not prime
9
10    // Base case 2: testVal only divisible by 1
11
12    // Recursive Case
13    ...
14    // Check if testVal can be evenly divided by divVal
15    // Hint: use the % operator
```

Run

CHALLENGE ACTIVITY

6.5.1: Creating a recursive function

5

Start

Write `MultiplySequence()`'s base case to output " = " and result if `inVal` is less than or equal to 2. End with a blank line.

Ex: If the input is 6, then the output is:

$$6 * 4 * 2 = 48$$

```
1 #include <iostream>
2 using namespace std;
3
4 void MultiplySequence(int inVal, int result) {
5     cout << inVal;
6     result = result * inVal;
7
8     /* Your code goes here */
9
10    else {
11        cout << " * ";
12        MultiplySequence(inVal - 2, result);
13    }
14 }
15 }
```

1

2

[Check](#)[Next level](#)

6.6 Recursive math functions

Fibonacci sequence

©zyBooks 06/06/23 23:52 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Recursive functions can solve certain math problems, such as computing the Fibonacci sequence. The **Fibonacci sequence** is 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, etc.; starting with 0, 1, the pattern is to compute the next number by adding the previous two numbers.

Below is a program that outputs the Fibonacci sequence values step-by-step, for a user-entered number of steps. The base case is that the program has output the requested number of steps. The recursive case is that the program needs to compute the number in the Fibonacci sequence.

Figure 6.6.1: Fibonacci sequence step-by-step.

©zyBooks 06/06/23 23:52 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

```
#include <iostream>
using namespace std;

/* Output the Fibonacci sequence step-by-step.
Fibonacci sequence starts as:
0 1 1 2 3 5 8 13 21 ... in which the first
two numbers are 0 and 1 and each additional
number is the sum of the previous two numbers
*/

void ComputeFibonacci(int fibNum1, int fibNum2, int
runCnt) {

    cout << fibNum1 << " + " << fibNum2 << " = "
        << fibNum1 + fibNum2 << endl;

    if (runCnt <= 1) { // Base case: Ran for user
specified
                    // number of steps, do nothing
    }
    else {          // Recursive case: compute next
value
        ComputeFibonacci(fibNum2, fibNum1 + fibNum2,
runCnt - 1);
    }
}

int main() {
    int runFor;      // User specified number of
values computed

    // Output program description
    cout << "This program outputs the" << endl
        << "Fibonacci sequence step-by-step," << endl
        << "starting after the first 0 and 1." << endl <<
endl;

    // Prompt user for number of values to compute
    cout << "How many steps would you like? ";
    cin >> runFor;

    // Output first two Fibonacci values, call
recursive function
    cout << "0" << endl << "1" << endl;
    ComputeFibonacci(0, 1, runFor);

    return 0;
}
```

@zyBooks 06/06/23 23:52 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

This program outputs the Fibonacci sequence step-by-step, starting after the first 0 and 1.

How many steps would you like? 10
0
1
0 + 1 = 1
1 + 1 = 2
1 + 2 = 3
2 + 3 = 5
3 + 5 = 8
5 + 8 = 13
8 + 13 = 21
13 + 21 = 34
21 + 34 = 55
34 + 55 = 89

zyDE 6.6.1: Recursive Fibonacci.

Complete ComputeFibonacci() to return F_N , where F_0 is 0, F_1 is 1, F_2 is 1, F_3 is 2, F_4 is 3, a continuing: F_N is $F_{N-1} + F_{N-2}$. Hint: Base cases are $N == 0$ and $N == 1$.

@zyBooks 06/06/23 23:52 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

```
1
2 #include <iostream>
3 using namespace std;
4
5 int ComputeFibonacci(int N) {
6
7     cout << "FIXME: Complete this function." << endl;
8     cout << "Currently just returns 0." << endl;
```

```
9      return 0;
10 }
11
12
13 int main() {
14     int N;          // F_N, starts at 0
15
16 }
```

Run

©zyBooks 06/06/23 23:52 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Greatest common divisor (GCD)

Recursion can solve the greatest common divisor problem. The **greatest common divisor** (GCD) is the largest number that divides evenly into two numbers, e.g. $\text{GCD}(12, 8) = 4$. One GCD algorithm (described by Euclid around 300 BC) subtracts the smaller number from the larger number until both numbers are equal. Ex:

- $\text{GCD}(12, 8)$: Subtract 8 from 12, yielding 4.
- $\text{GCD}(4, 8)$: Subtract 4 from 8, yielding 4.
- $\text{GCD}(4, 4)$: Numbers are equal, return 4

The following recursively computes the GCD of two numbers. The base case is that the two numbers are equal, so that number is returned. The recursive case subtracts the smaller number from the larger number and then calls GCD with the new pair of numbers.

Figure 6.6.2: Calculate greatest common divisor of two numbers.

©zyBooks 06/06/23 23:52 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

```
#include <iostream>
using namespace std;

/* Determine the greatest common divisor
   of two numbers, e.g. GCD(8, 12) = 4
*/
int GCDcalculator(int inNum1, int inNum2) {
    int gcdVal; // Holds GCD results

    if(inNum1 == inNum2) { // Base case: Numbers are equal
        gcdVal = inNum1; // Return value
    }
    else { // Recursive case: subtract smaller from larger
        if (inNum1 > inNum2) { // Call function with new values
            gcdVal = GCDcalculator(inNum1 - inNum2,
inNum2);
        }
        else {
            gcdVal= GCDcalculator(inNum1, inNum2 -
inNum1);
        }
    }

    return gcdVal;
}

int main() {
    int gcdInput1; // First input to GCD calc
    int gcdInput2; // Second input to GCD calc
    int gcdOutput; // Result of GCD

    // Print program function
    cout << "Program outputs the greatest \n"
        << "common divisor of two numbers." << endl;

    // Prompt user for input
    cout << "Enter first number: ";
    cin >> gcdInput1;

    cout << "Enter second number: ";
    cin >> gcdInput2;

    // Check user values are > 1, call recursive GCD
    // function
    if ((gcdInput1 < 1) || (gcdInput2 < 1)) {
        cout << "Note: Neither value can be below 1." <<
endl;
    }
    else {
        gcdOutput = GCDcalculator(gcdInput1, gcdInput2);
        cout << "Greatest common divisor = " <<
gcdOutput << endl;
    }

    return 0;
}
```

@zyBooks 06/06/23 23:52 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

Program outputs the greatest common divisor of two numbers.
 Enter first number: 12
 Enter second number: 8
 Greatest common divisor = 4

...

Program outputs the greatest common divisor of two numbers.
 Enter first number: 456
 Enter second number:
 784
 Greatest common divisor = 8

...

Program outputs the greatest common divisor of two numbers.
 Enter first number: 0
 Enter second number: 10
 Note: Neither value can be below 1.

@zyBooks 06/06/23 23:52 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

6.6.1: Recursive GCD example.



- 1) How many calls are made to GCDcalculator() function for input values 12 and 8?



- 1
- 2
- 3

2) What is the base case for the GCD algorithm?

- When both inputs to the function are equal.
- When both inputs are greater than 1.
- When $\text{inNum1} > \text{inNum2}$.

©zyBooks 06/06/23 23:52 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Exploring further:

- [Fibonacci number](#) from Wolfram.
- [Greatest Common Divisor](#) from Wolfram.

CHALLENGE ACTIVITY

6.6.1: Writing a recursive math function.



Write code to complete `RaiseToPower()`. Sample output if `userBase` is 4 and `userExponent` is 2 is shown below. Note: This example is for practicing recursion; a non-recursive function, or using the built-in function `pow()`, would be more common.

$4^2 = 16$

[Learn how our autograder works](#)

489394.3384924.qx3zqy7

```
1 #include <iostream>
2 using namespace std;
3
4 int RaiseToPower(int baseVal, int exponentVal){
5     int resultVal;
6
7     if (exponentVal == 0) {
8         resultVal = 1;
9     }
10    else {
11        resultVal = baseVal * /* Your solution goes here */;
12    }
13
14    return resultVal;
15 }
```

©zyBooks 06/06/23 23:52 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Run

View your last submission ▾

6.7 Recursive exploration of all possibilities

Recursion is a powerful technique for exploring all possibilities, such as all possible reorderings of a word's letters, all possible subsets of items, all possible paths between cities, etc. This section provides several examples.

Word scramble

Consider printing all possible combinations (or "scramblings") of a word's letters. The letters of abc can be scrambled in 6 ways: abc, acb, bac, bca, cab, cba. Those possibilities can be listed by making three choices: Choose the first letter (a, b, or c), then choose the second letter, then choose the third letter. The choices can be depicted as a tree. Each level represents a choice. Each node in the tree shows the unchosen letters on the left, and the chosen letters on the right.

PARTICIPATION
ACTIVITY

6.7.1: Exploring all possibilities viewed as a tree of choices.



Animation content:

Static figure: A tree of choices with 4 levels and 16 nodes. Each node contains the remaining letters and chosen letters separated with a /. The root node contains abc/ and has 3 children. The second level containing 3 nodes has the label "Choose first letter". From left to right, the nodes contain bc/a, ac/b, and ab/c, respectively. Each node on the second level has 2 children. The third level containing 6 nodes has the label "Choose second letter". From left to right, the nodes contain c/ab, b/ac, c/ba, a/bc, b/ca, and a/cb, respectively. Each node on the third level has 1 child. The fourth level containing 6 nodes has the label "Choose third letter". From left to right, the nodes contain /abc, /acb, /bac, /bca, /cab, and /cba, respectively.

Animation captions:

1. Consider printing all possible combinations of a word's letters. Those possibilities can be listed by choosing the first letter, then the second letter, then the third letter.
2. The choices can be depicted as a tree. Each level represents a choice.
3. A recursive exploration function is a natural match to print all possible combinations of a string's letters. Each call to the function chooses from the set of unchosen letters, continuing until no unchosen letters remain.

The tree guides creation of a recursive exploration function to print all possible combinations of a string's letters. The function takes two parameters: unchosen letters, and already chosen letters. The base case is no unchosen letters, causing printing of the chosen letters. The recursive case calls the function once for each letter in the unchosen letters. The above animation depicts how the recursive algorithm traverses the tree. The tree's leaves (the bottom nodes) are the base cases.

The following program prints all possible ordering of the letters of a user-entered word.

Figure 6.7.1: Scramble a word's letters in every possible way.

©zyBooks 06/06/23 23:52 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

```
#include <iostream>
#include <string>
using namespace std;

/* Output every possible combination of a word.
   Each recursive call moves a letter from
   remainLetters to scramLetters.
*/
void ScrambleLetters(string remainLetters, // Remaining
                      letters
                      string scramLetters) { // Scrambled
    letters
    string tmpString; // Temp word combination
    unsigned int i; // Loop index

    if (remainLetters.size() == 0) { // Base case: All
        letters used
        cout << scramLetters << endl;
    }
    else { // Recursive case:
        move a letter from
        // remaining to
        scrambled letters
        for (i = 0; i < remainLetters.size(); ++i) {
            // Move letter to scrambled letters
            tmpString = remainLetters.substr(i, 1);
            remainLetters.erase(i, 1);
            scramLetters = scramLetters + tmpString;

            ScrambleLetters(remainLetters, scramLetters);

            // Put letter back in remaining letters
            remainLetters.insert(i, tmpString);
            scramLetters.erase(scramLetters.size() - 1, 1);
        }
    }
}

int main() {
    string wordScramble; // User defined word to scramble

    // Prompt user for input
    cout << "Enter a word to be scrambled: ";
    cin >> wordScramble;

    // Call recursive function
    ScrambleLetters(wordScramble, "");

    return 0;
}
```

©zyBooks 06/06/23 23:52 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

Enter a word to be
 scrambled: cat
 cat
 cta
 act
 atc
 tca
 tac

PARTICIPATION ACTIVITY

6.7.2: Letter scramble.



- 1) What is the output of
 ScrambleLetters("xy", "")? Determine
 your answer by manually tracing the
 code, not by running the program.

- xy xy
- xx yy xy yx
- xy yx

©zyBooks 06/06/23 23:52 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

Shopping spree

Recursion can find all possible subsets of a set of items. Consider a shopping spree in which a person can select any 3-item subset from a larger set of items. The following program prints all possible 3-item subsets of a given larger set. The program also prints the total price of each subset.

ShoppingBagCombinations() has a parameter for the current bag contents, and a parameter for the remaining items from which to choose. The base case is that the current bag already has 3 items, which prints the items. The recursive case moves one of the remaining items to the bag, recursively calling the function, then moving the item back from the bag to the remaining items.

©zyBooks 06/06/23 23:52 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Figure 6.7.2: Shopping spree in which a user can fit 3 items in a shopping bag.

©zyBooks 06/06/23 23:52 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

```
Milk   Belt
Toys  = $45
Milk   Belt
Cups   = $38
Milk   Toys
Belt   = $45
Milk   Toys
Cups   = $33
Milk   Cups
Belt   = $38
Milk   Cups
Toys   = $33
Belt   Milk
Toys   = $45
Belt   Milk
Cups   = $38
Belt   Toys
Milk   = $45
Belt   Toys
Cups   = $55
Belt   Cups
Milk   = $38
Belt   Cups
Toys   = $55
Toys   Milk
Belt   = $45
Toys   Milk
Cups   = $33
Toys   Belt
Milk   = $45
Toys   Belt
Cups   = $55
Toys   Cups
Milk   = $33
Toys   Cups
Belt   = $55
Cups   Milk
Belt   = $38
Cups   Milk
Toys   = $33
Cups   Belt
Milk   = $38
Cups   Belt
Toys   = $55
Cups   Toys
Milk   = $33
Cups   Toys
Belt   = $55
```

zyBooks 06/06/23 23:52 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

©zyBooks 06/06/23 23:52 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

class Item {
public:
    string itemName; // Name of item
    int priceDollars; // Price of item
};

const unsigned int MAX_ITEMS_IN_SHOPPING_BAG = 3; // Max num items

/* Output every combination of items that fit
   in a shopping bag. Each recursive call moves
   one item into the shopping bag.
*/
void ShoppingBagCombinations(vector<Item> currBag,           // Bag contents
                               vector<Item> remainingItems) { // Available items
    int bagValue;           // Cost of items in shopping bag
    Item tmpGroceryItem; // Grocery item to add to bag
    unsigned int i;         // Loop index

    if (currBag.size() == MAX_ITEMS_IN_SHOPPING_BAG) { // Base case: Shopping bag full
        bagValue = 0;
        for (i = 0; i < currBag.size(); ++i) {
            bagValue += currBag.at(i).priceDollars;
            cout << currBag.at(i).itemName << " ";
        }
        cout << "= $" << bagValue << endl;
    } else { // Recursive case: move one
        for (i = 0; i < remainingItems.size(); ++i) { // item to bag
            // Move item into bag
            tmpGroceryItem = remainingItems.at(i);
            remainingItems.erase(remainingItems.begin() + i);
            currBag.push_back(tmpGroceryItem);

            ShoppingBagCombinations(currBag, remainingItems);

            // Take item out of bag
            remainingItems.insert(remainingItems.begin() +
i, tmpGroceryItem);
            currBag.pop_back();
        }
    }
}

int main() {
    vector<Item> possibleItems(0); // Possible shopping items
    vector<Item> shoppingBag(0); // Current shopping bag
    Item tmpGroceryItem; // Temp item

    // Populate grocery with different items
    tmpGroceryItem.itemName = "Milk";
    tmpGroceryItem.priceDollars = 2;
    possibleItems.push_back(tmpGroceryItem);

    tmpGroceryItem.itemName = "Belt";
    tmpGroceryItem.priceDollars = 24;
    possibleItems.push_back(tmpGroceryItem);

    tmpGroceryItem.itemName = "Toys";
    tmpGroceryItem.priceDollars = 19;
    possibleItems.push_back(tmpGroceryItem);

    tmpGroceryItem.itemName = "Gum";
}
```

©zyBooks 06/06/23 23:52 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

©zyBooks 06/06/23 23:52 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

**PARTICIPATION
ACTIVITY**

6.7.3: All letter combinations.



- 1) When main() calls ShoppingBagCombinations(), how many items are in the remainingItems list?

- None
- 3
- 4

©zyBooks 06/06/23 23:52 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 2) When main() calls ShoppingBagCombinations(), how many items are in currBag list?

- None
- 1
- 4

- 3) After main() calls ShoppingBagCombinations(), what happens first?

- The base case prints Milk, Belt, Toys.
- The function bags one item, makes recursive call.
- The function bags 3 items, makes recursive call.

- 4) Just before ShoppingBagCombinations() returns back to main(), how many items are in the remainingItems list?

- None
- 4

- 5) How many recursive calls occur before the first combination is printed?

- None
- 1
- 3

©zyBooks 06/06/23 23:52 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 6) What happens if main() only put 2, rather than 4, items in the possibleItems list?

- Base case never executes; nothing printed.
- Infinite recursion occurs.



Traveling salesman

Recursion is useful for finding all possible paths. Suppose a salesman must travel to 3 cities: Boston, Chicago, and Los Angeles. The salesman wants to know all possible paths among those three cities, starting from any city. A recursive exploration of all travel paths can be used. The base case is that the salesman has traveled to all cities. The recursive case is to travel to a new city, explore possibilities, then return to the previous city.

Figure 6.7.3: Find distance of traveling to 3 cities.

©zyBooks 06/06/23 23:52 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

©zyBooks 06/06/23 23:52 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Boston	Chicago	Los
Angeles	= 2971	
Boston	Los Angeles	
Chicago	= 4971	
Chicago	Boston	Los
Angeles	= 3920	
Chicago	Los Angeles	
Boston	= 4971	
Los Angeles	Boston	
Chicago	= 3920	
Los Angeles	Chicago	
Boston	= 2971	

@zyBooks 06/06/23 23:52 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

@zyBooks 06/06/23 23:52 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

```

#include <iostream>
#include <iomanip>
#include <vector>
using namespace std;

const unsigned int NUM_CITIES = 3;           // Number of
cities
int cityDistances[NUM_CITIES][NUM_CITIES]; // Distance
between cities
string cityNames[NUM_CITIES];               // City
names

/* Output every possible travel path.
   Each recursive call moves to a new city.
*/
void TravelPaths(vector<int> currPath, vector<int>
needToVisit) {
    int totalDist;      // Total distance given current
path
    int tmpCity;        // Next city distance
    unsigned int i;     // Loop index

    if (currPath.size() == NUM_CITIES) { // Base case:
Visited all cities
        totalDist = 0;                  // return
total path distance
        for (i = 0; i < currPath.size(); ++i) {
            cout << cityNames[currPath.at(i)] << "    ";

            if (i > 0) {
                totalDist += cityDistances[currPath.at(i - 1)][currPath.at(i)];
            }
        }

        cout << "= " << totalDist << endl;
    }
    else {                      // Recursive
case: pick next city
        for (i = 0; i < needToVisit.size(); ++i) {
            // Add city to travel path
            tmpCity = needToVisit.at(i);
            needToVisit.erase(needToVisit.begin() + i);
            currPath.push_back(tmpCity);

            TravelPaths(currPath, needToVisit);

            // Remove city from travel path
            needToVisit.insert(needToVisit.begin() + i,
tmpCity);
            currPath.pop_back();
        }
    }
}

int main() {
    vector<int> needToVisit(0); // Cities left to visit
    vector<int> currPath(0);   // Current path traveled

    // Initialize distances array
    cityDistances[0][0] = 0;
    cityDistances[0][1] = 960; // Boston-Chicago
    cityDistances[0][2] = 2960; // Boston-Los Angeles
    cityDistances[1][0] = 960; // Chicago-Boston
    cityDistances[1][1] = 0;
    cityDistances[1][2] = 2011; // Chicago-Los Angeles
    cityDistances[2][0] = 2960; // Los Angeles-Boston
    cityDistances[2][1] = 2011; // Los Angeles-Chicago
    cityDistances[2][2] = 0;

    cityNames[0] = "Boston";
    cityNames[1] = "Chicago";
    cityNames[2] = "Los Angeles";
}

```

©zyBooks 06/06/23 23:52 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

©zyBooks 06/06/23 23:52 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

6.7.4: Recursive exploration.



1) You wish to generate all possible 3-letter subsets from the letters in an N-letter word ($N > 3$). Which of the above recursive functions is the closest?

- ShoppingBagCombinations
- ScrambleLetters
- main()

©zyBooks 06/06/23 23:52 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

CHALLENGE ACTIVITY

6.7.1: Enter the output of recursive exploration.



489394.3384924.qx3zqy7

Start

Type the program's output

```
#include <iostream>
#include <vector>
using namespace std;

void ScrambleNums(vector<int> remainNums, vector<int> scramNums) {
    vector<int> tmpRemainNums;
    int tmpRemovedNum;
    int i;

    if (remainNums.size() == 0) {
        cout << scramNums.at(0);
        cout << scramNums.at(1);
        cout << scramNums.at(2) << endl;
    }
    else {
        for (i = 0; i < remainNums.size(); ++i) {
            tmpRemainNums = remainNums; // Make a copy.
            tmpRemovedNum = tmpRemainNums.at(i);
            tmpRemainNums.erase(tmpRemainNums.begin() + i); // Remove element at i
            scramNums.push_back(tmpRemovedNum);
            ScrambleNums(tmpRemainNums, scramNums);
            scramNums.erase(scramNums.end() - 1); // Remove last element
        }
    }
}

int main() {
    vector<int> numsToScramble;
    vector<int> resultNums;

    numsToScramble.push_back(2);
    numsToScramble.push_back(9);
    numsToScramble.push_back(0);

    ScrambleNums(numsToScramble, resultNums);

    return 0;
}
```

©zyBooks 06/06/23 23:52 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

1

2

Check**Next**

**CHALLENGE
ACTIVITY**

6.7.2: Recursive exploration of all possibilities.



489394.3384924.qx3zqy7

Start

Integer vectSize is read from input, then vectSize strings are read and stored into vector passengersTo write the base case to output each element in vector pickedPassengers if the size of vector remainPas space after each element. End with a newline.

@zyBooks 06/06/23 23:52 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Ex: If the input is:

```
3
Kim Rob Zoe
```

then the output is:

```
All unique queues:
Kim Rob Zoe
Kim Zoe Rob
Rob Kim Zoe
Rob Zoe Kim
Zoe Kim Rob
Zoe Rob Kim
```

```
1 #include <iostream>
2 #include <vector>
3 using namespace std;
4
5 void OrderPassengers(vector<string> remainPassengers, vector<string> pickedPass
6     unsigned int i;
7     string pick;
8
9     /* Your code goes here */
10
11    else {
12        for (i = 0; i < remainPassengers.size(); ++i) {
13            pick = remainPassengers.at(i);
14            remainPassengers.erase(remainPassengers.begin() + i);
15            pickedPassengers.push_back(pick);
```

1

2

Check**Next level**

@zyBooks 06/06/23 23:52 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Exploring further:

- [Recursive Algorithms](#) from khanacademy.org

6.8 Stack overflow

Recursion enables an elegant solution to some problems. But, for large problems, deep recursion can cause memory problems. Part of a program's memory is reserved to support function calls. Each function call places a new **stack frame** on the stack, for local parameters, local variables, and more function items. Upon return, the frame is deleted.

Deep recursion could fill the stack region and cause a **stack overflow**, meaning a stack frame extends beyond the memory region allocated for stack. Stack overflow usually causes the program to crash and report an error like: segmentation fault, access violation, or bad access.

PARTICIPATION ACTIVITY
6.8.1: Recursion causing stack overflow.
 ©zyBooks 06/06/23 23:52 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Animation captions:

1. Deep recursion may cause stack overflow, causing a program to crash.

The animation showed a tiny stack region for easy illustration of stack overflow.

The size of parameters and local variables results in a larger stack frame. Large vectors, arrays, or strings declared as local variables, or passed by copy, can lead to faster stack overflow.

A programmer can estimate recursion depth and stack size to determine whether stack overflow might occur. Sometimes a non-recursive algorithm must be developed to avoid stack overflow.

PARTICIPATION ACTIVITY
6.8.2: Stack overflow.


- 1) A memory's stack region can store at most one stack frame.

- True
- False

- 2) The size of the stack is unlimited.

- True
- False

- 3) A stack overflow occurs when the stack frame for a function call extends past the end of the stack's memory.

- True
- False

- 4) The following recursive function will result in a stack overflow.

```
int RecAdder(int inValue) {
    return RecAdder(inValue + 1);
}
```

 ©zyBooks 06/06/23 23:52 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

- True
- False

6.9 C++ example: Recursively output permutations

zyDE 6.9.1: Recursively output permutations.

The below program prints all permutations of an input string of letters, one permutation per line. Ex: The six permutations of "cab" are:

```
cab  
cba  
acb  
abc  
bca  
bac
```

©zyBooks 06/06/23 23:52 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Below, the PermuteString function works recursively by starting with the first character and permuting the remainder of the string. The function then moves to the second character and permutes the string consisting of the first character and the third through the end of the string, and so on.

1. Run the program and input the string "cab" (without quotes) to see that the above output is produced.
2. Modify the program to print the permutations in the opposite order, and also to output the permutation count on each line.
3. Run the program again and input the string cab. Check that the output is reversed.
4. Run the program again with an input string of abcdef. Why did the program take longer to produce the results?

[Load default template](#)

```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 // FIXME: Use a static variable to count permutations. Why should this be?
6
7 void PermuteString(string head, string tail) {
8     char current;
9     string newPermute;
10    int len;
11    int i;
12
13    current = '?';
14
15    len = tail.size();
16    for (i = 0; i < len; i++) {
17        newPermute = head + tail[i];
18        tail = tail.substr(0, i) + tail.substr(i+1);
19        PermuteString(newPermute, tail);
20    }
21}
```

cab

[Run](#)

©zyBooks 06/06/23 23:52 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

zyDE 6.9.2: Recursively output permutations (solution).

Below is the solution to the above problem.

[Load default template](#)

```

1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 static int permutationCount = 0;
6
7 void PermuteString(string head, string tail) {
8     char current;
9     string newPermute;
10    int len;
11    int i;
12
13    current = '?';
14
15    len = tail.size();
16    if (len == 0) {
17        cout << head << endl;
18        permutationCount++;
19    } else {
20        for (i = 0; i < len; i++) {
21            current = tail[i];
22            tail.erase(i, 1);
23            newPermute = head + current + tail;
24            PermuteString(newPermute, tail);
25            tail.insert(i, 1, current);
26        }
27    }
28}
```

©zyBooks 06/06/23 23:52 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

cab
abcdef

[Run](#)

6.10 Recursive definitions



This section has been set as optional by your instructor.

Recursive algorithms

An **algorithm** is a sequence of steps, including at least 1 terminating step, for solving a problem. A **recursive algorithm** is an algorithm that breaks the problem into smaller subproblems and applies the algorithm itself to solve the smaller subproblems.

Because a problem cannot be endlessly divided into smaller subproblems, a recursive algorithm must have a **base case**: A case where a recursive algorithm completes without applying itself to a smaller subproblem.

PARTICIPATION
ACTIVITY

6.10.1: Recursive factorial algorithm for positive numbers.



©zyBooks 06/06/23 23:52 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Animation captions:

1. A recursive algorithm to compute N factorial has 2 parts: the base case and the non-base case.
2. N is assumed to be a positive integer. The base case is when N equals 1, wherein a result of 1 is returned.
3. The non-base case computes the result by multiplying N by (N - 1) factorial.
4. The algorithm applying itself to a smaller subproblem is what makes the algorithm recursive.

PARTICIPATION ACTIVITY

6.10.2: Recursive algorithms.



- 1) A recursive algorithm applies itself to a smaller subproblem in all cases.

True
 False

- 2) The base case is what ensures that a recursive algorithm eventually terminates.

True
 False

- 3) The presence of a base case is what identifies an algorithm as being recursive.

True
 False

©zyBooks 06/23 23:52 1692462

 Taylor Larrechea
 COLORADOCSPB2270Summer2023

Recursive functions

A **recursive function** is a function that calls itself. Recursive functions are commonly used to implement recursive algorithms.

Table 6.10.1: Sample recursive functions: Factorial, CumulativeSum, and ReverseList.

```
Factorial(N) {
    if (N == 1)
        return 1
    else
        return N * Factorial(N - 1)
}
```

```
CumulativeSum(N) {
    if (N == 0)
        return 0
    else
        return N + CumulativeSum(N - 1)
}
```

```
ReverseList(list, startIndex, endIndex) {
    if (startIndex >= endIndex)
        return
    else {
        Swap elements at startIndex and endIndex
        ReverseList(list, startIndex + 1, endIndex - 1)
    }
}
```

©zyBooks 06/06/23 23:52 1692462

 Taylor Larrechea
 COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

6.10.3: CumulativeSum recursive function.



- 1) What is the condition for the base case in the CumulativeSum function?

N equals 0

- N does not equal 0
- 2) If Factorial(6) is called, how many additional calls are made to Factorial to compute the result of 720?
- 7
- 5
- 3
- 3) Suppose ReverseList is called on a list of size 3, a start index of 0, and an out-of-bounds end index of 3. The base case ensures that the function still properly reverses the list.
- True
- False

@zyBooks 06/06/23 23:52 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

6.11 Recursive algorithms



This section has been set as optional by your instructor.

Fibonacci numbers

The **Fibonacci sequence** is a numerical sequence where each term is the sum of the previous 2 terms in the sequence, except the first 2 terms, which are 0 and 1. A recursive function can be used to calculate a **Fibonacci number**: A term in the Fibonacci sequence.

Figure 6.11.1: FibonacciNumber recursive function.

```
FibonacciNumber(termIndex) {  
    if (termIndex == 0)  
        return 0  
    else if (termIndex == 1)  
        return 1  
    else  
        return FibonacciNumber(termIndex - 1) + FibonacciNumber(termIndex  
- 2)  
}
```

@zyBooks 06/06/23 23:52 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

6.11.1: FibonacciNumber recursive function.

- 1) What does FibonacciNumber(2) return?

/

Check

Show answer

- 2) What does FibonacciNumber(4)
return?

Check**Show answer**

- 3) What does FibonacciNumber(8)
return?

Check**Show answer**

©zyBooks 06/06/23 23:52 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Recursive binary search

Binary search is an algorithm that searches a sorted list for a key by first comparing the key to the middle element in the list and recursively searching half of the remaining list so long as the key is not found.

Binary search first checks the middle element of the list. If the search key is found, the algorithm returns the index. If the search key is not found, the algorithm recursively searches the remaining left sublist (if the search key was less than the middle element) or the remaining right sublist (if the search key was greater than the middle element).

Figure 6.11.2: BinarySearch recursive algorithm.

```
BinarySearch(numbers, low, high, key) {
    if (low > high)
        return -1

    mid = (low + high) / 2
    if (numbers[mid] < key) {
        return BinarySearch(numbers, mid + 1, high,
key)
    }
    else if (numbers[mid] > key) {
        return BinarySearch(numbers, low, mid - 1,
key)
    }
    return mid
}
```

PARTICIPATION
ACTIVITY

6.11.2: Recursive binary search.



Suppose $\text{BinarySearch}(\text{numbers}, 0, 6, 42)$ is used to search the list (14, 26, 42, 59, 71, 88, 92).
for key 42.

©zyBooks 06/06/23 23:52 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 1) What is the first middle element that is
compared against 42?



- 42
- 59
- 71



2) What will the low and high argument values be for the first recursive call?

- low = 0
high = 2
- low = 0
high = 3
- low = 4
high = 6

3) How many calls to BinarySearch will be made by the time 42 is found?

- 2
- 3
- 4

©zyBooks 06/06/23 23:52 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



PARTICIPATION ACTIVITY

6.11.3: Recursive binary search base case.



1) Which does not describe a base case for BinarySearch?

- The low argument is greater than the high argument.
- The list element at index `mid` equals the key.
- The list element at index `mid` is less than the key.

©zyBooks 06/06/23 23:52 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

7.1 Binary trees

Binary tree basics

In a list, each node has up to one successor. In a **binary tree**, each node has up to two children, known as a *left child* and a *right child*. "Binary" means two, referring to the two children. Some more definitions related to a binary tree:

- **Leaf**: A tree node with no children.
- **Internal node**: A node with at least one child.
- **Parent**: A node with a child is said to be that child's parent. A node's **ancestors** include the node's parent, the parent's parent, etc., up to the tree's root.
- **Root**: The one tree node with no parent (the "top" node).

Another section discusses binary tree usefulness; this section introduces definitions.

Below, each node is represented by just the node's key, as in B, although the node may have other data.

PARTICIPATION
ACTIVITY

7.1.1: Binary tree basics.



Animation captions:

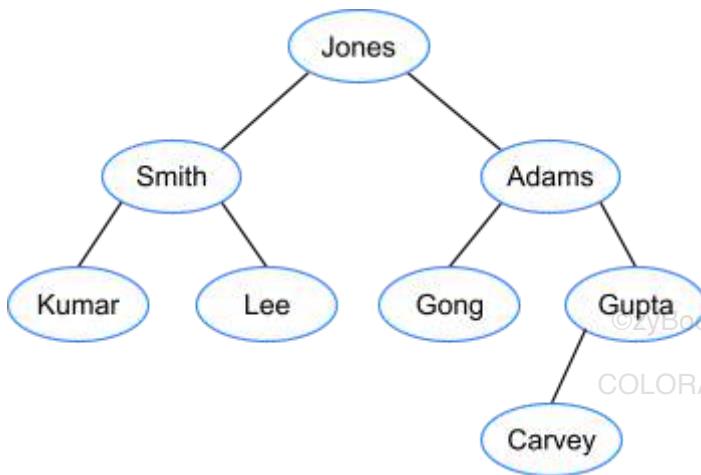
1. In a list, each node has up to one successor.
2. In a binary tree, each node has up to two children.
3. A tree is normally drawn vertically. Edge arrows are optional.
4. A node can have a left child and right child. A node with a child is called the child's parent.
5. First node: Root node. Node without child: Leaf node. Node with child: Internal nodes.

PARTICIPATION
ACTIVITY

7.1.2: Binary tree basics.



©zyBooks 06/06/23 23:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

- 1) Root node: _____.

 //**Check****Show answer**

- 2) Smith's left child: _____.

 //**Check****Show answer**

- 3) The tree has four leaf nodes:

Kumar, Lee, Gong, and _____.

 //**Check****Show answer**

- 4) How many internal nodes does the tree have?

 //**Check****Show answer**

- 5) The tree has an internal node, Gupta, with only one child rather than two. Is the tree still a binary tree? Type yes or no.

 //

©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Check**Show answer**

Depth, level, and height

A few additional terms:

- The link from a node to a child is called an **edge**.
- A node's **depth** is the number of edges on the path from the root to the node. The root node thus has depth 0.
- All nodes with the same depth form a tree **level**.
- A tree's **height** is the largest depth of any node. A tree with just one node has height 0.

©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

PARTICIPATION ACTIVITY

7.1.3: Binary tree terminology: height, depth, and level.

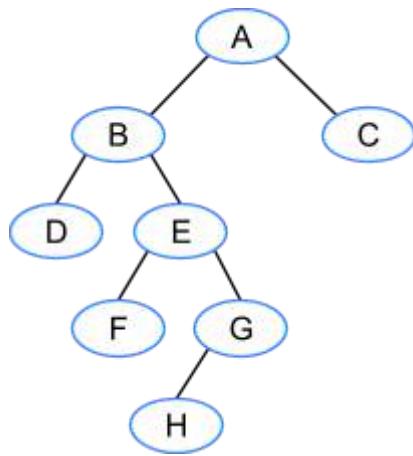


Animation captions:

1. The binary tree has edges A to B, A to C, and C to D.
2. A node's depth is the number of edges from the root to the node.
3. Nodes with the same depth form a level.
4. A tree's height is the largest depth of any node.

PARTICIPATION ACTIVITY

7.1.4: Binary tree height, depth, and level.



©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

1) A's depth is 1.

- True
 False

2) E's depth is 2.

- True

False

3) B and C form level 1.

 True False

4) D, F, and H form level 2.

©zyBooks 06/06/23 23:54 169246

Taylor Larrechea

COLORADOCSPB2270Summer2023

 True False

5) The tree's height is 4.

 True False

6) A tree with just one node has height 0.

 True False

Special types of binary trees

Certain binary tree structures can affect the speed of operations on the tree. The following describe special types of binary trees:

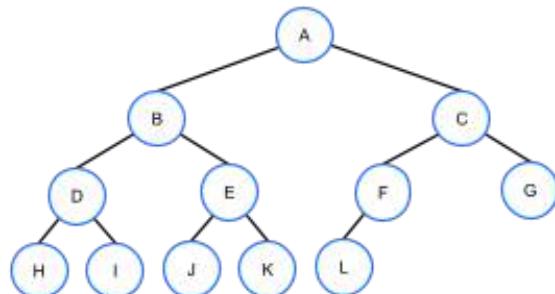
- A binary tree is **full** if every node contains 0 or 2 children.
- A binary tree is **complete** if all levels, except possibly the last level, contain all possible nodes and all nodes in the last level are as far left as possible.
- A binary tree is **perfect**, if all internal nodes have 2 children and all leaf nodes are at the same level.

Figure 7.1.1: Special types of binary trees: full, complete, perfect.

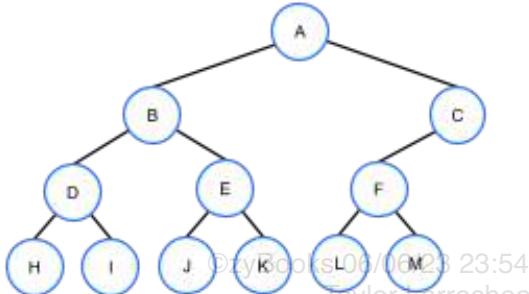
©zyBooks 06/06/23 23:54 169246

Taylor Larrechea

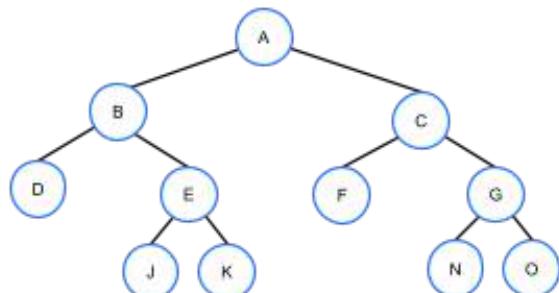
COLORADOCSPB2270Summer2023



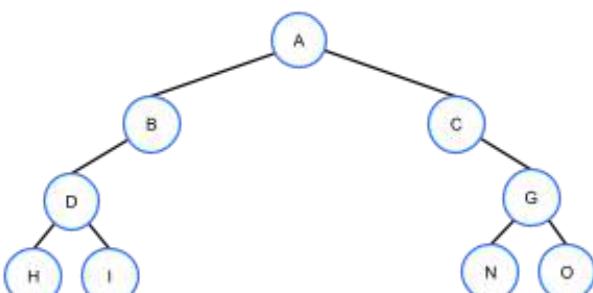
Not full, complete, not perfect

©zyBooks 06/06/23 23:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

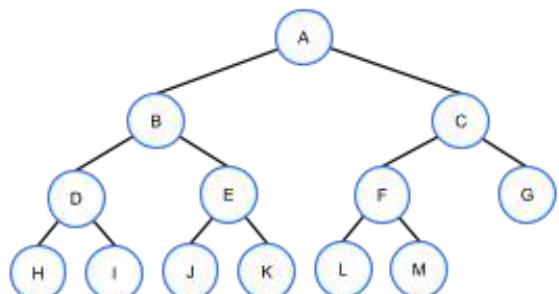
Not full, not complete, not perfect



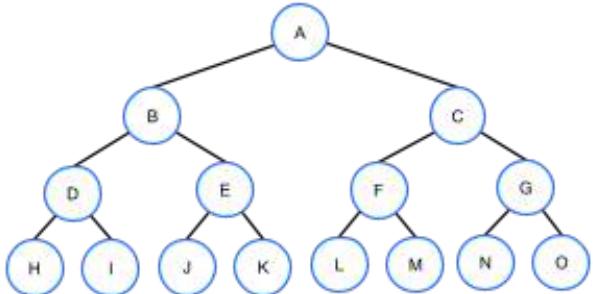
Full, not complete, not perfect



Not full, not complete, not perfect



Full, complete, not perfect

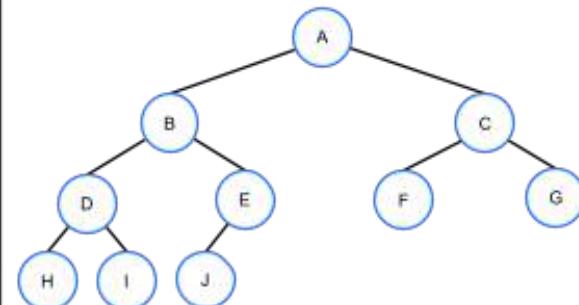
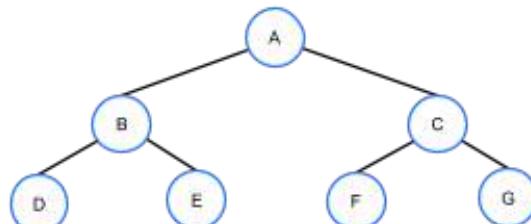
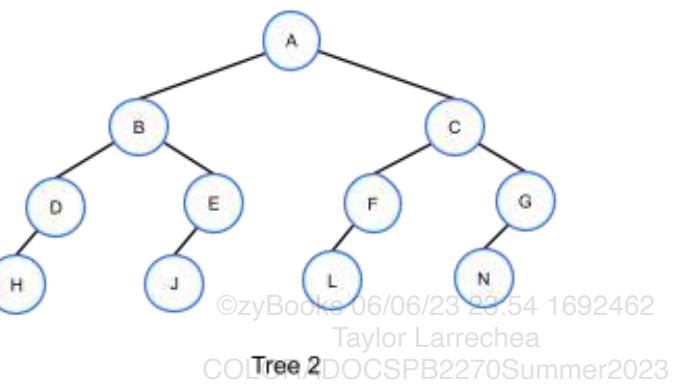
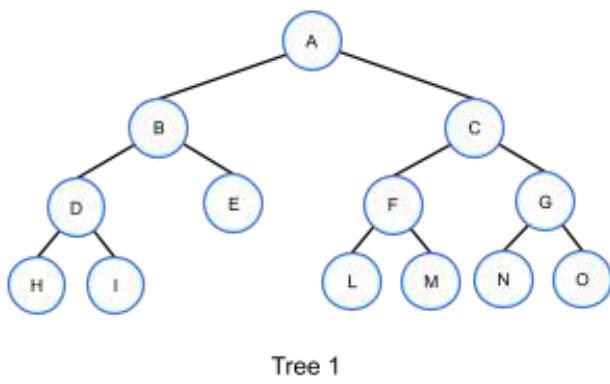


Full, complete, perfect

PARTICIPATION ACTIVITY

7.1.5: Identifying special types of binary trees.





If unable to drag and drop, refresh the page.

Not full, not complete, not perfect

Full, complete, perfect

Full, not complete, not perfect

Not full, complete, not perfect

Tree 1

Tree 2

Tree 3

Tree 4

©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea
COLORADOCSPB2270Summer2023

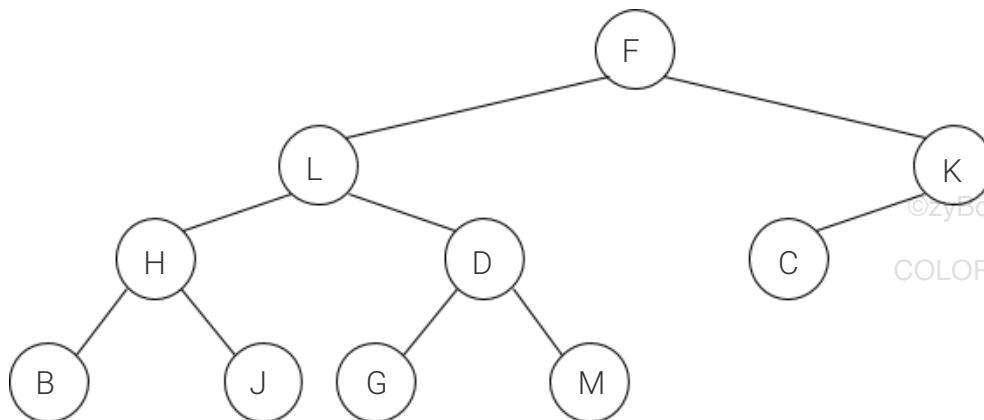
Reset

CHALLENGE ACTIVITY

7.1.1: Binary trees.



489394.3384924.qx3zqy7

Start

©zyBooks 06/06/23 23:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

What is the root node?

Ex: A

What is the parent of node J?

Ex: A

What are the ancestors of node J?

Ex: A, B, C

What are the leaf nodes?

Ex: A, B, C

How many internal nodes does the tree have?

Ex: 9

1

2

3

Check**Next**

7.2 Applications of trees

File systems

Trees are commonly used to represent hierarchical data. A tree can represent files and directories in a file system, since a file system is a hierarchy.

©zyBooks 06/06/23 23:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

7.2.1: A file system is a hierarchy that can be represented by a tree.



Animation content:

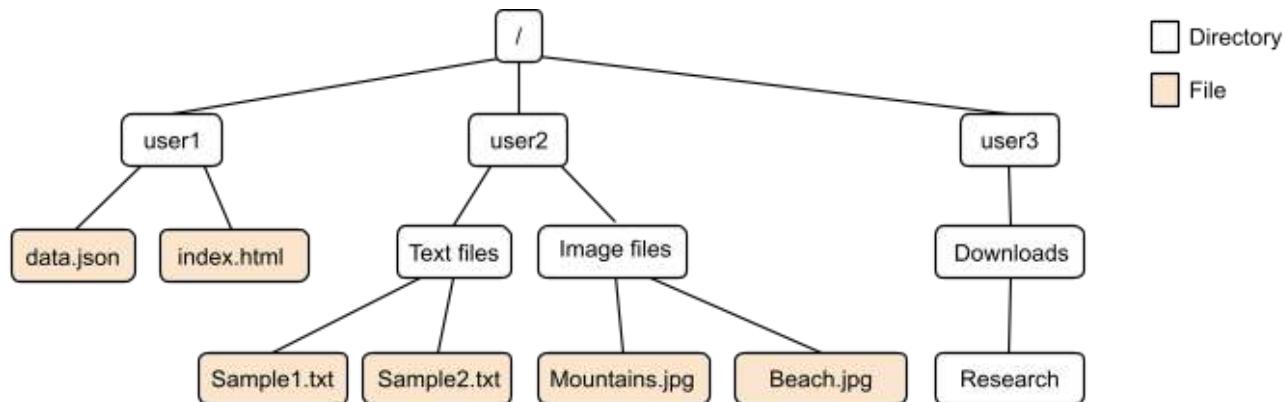
undefined

Animation captions:

1. A tree representing a file system has the filesystem's root directory ("/"), represented by the root node.
 2. The root contains 2 configuration text files and 2 additional directories: user1 and user2.
 3. Directories contain additional entries. Only empty directories will be leaf nodes. All files are leaf nodes.
- ©zyBooks 06/06/23 23:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

7.2.2: Analyzing a file system tree.



- Directory
- File

1) What is the depth of the "Mountains.jpg" file node?



- 3
- 4

2) What is the tree's height?



- 3
- 4
- 14

3) What is the parent of the "Text files" node?



- The "user2" directory node
- The "Image files" directory node
- The "Sample1.txt" file node.

4) Which operation would increase the height of the tree?



©zyBooks 06/06/23 23:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- Adding a new file into the user1 directory
- Adding a new directory into the "Image files" directory
- Adding a new directory into the "Research" directory

©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

7.2.3: File system trees.



- 1) A file in a file system tree is always a leaf node.
 - True
 - False
- 2) A directory in a file system tree is always an internal node.
 - True
 - False
- 3) Using a tree data structure to implement a file system requires that each directory node support a variable number of children.
 - True
 - False



Binary space partitioning

Binary space partitioning (BSP) is a technique of repeatedly separating a region of space into 2 parts and cataloging objects contained within the regions. A **BSP tree** is a binary tree used to store information for binary space partitioning. Each node in a BSP tree contains information about a region of space and which objects are contained in the region.

©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

In graphics applications, a BSP tree can be used to store all objects in a multidimensional world. The BSP tree can then be used to efficiently determine which objects must be rendered on screen. The viewer's position in space is used to perform a lookup within the BSP tree. The lookup quickly eliminates a large number of objects that are not visible and therefore should not be rendered.

PARTICIPATION ACTIVITY

7.2.4: A BSP tree is used to quickly determine which objects do not need to be rendered.



Animation content:

undefined

Animation captions:

1. Data for a large, open 2-D world contains many objects. Only a few objects are visible on screen at any given moment.
2. Avoiding rendering off-screen objects is crucial for realtime graphics. But checking the intersection of all objects with the screen's rectangle is too time consuming.
3. A BSP tree represents partitioned space. The root represents the entire world and stores a list of all objects in the world, as well as the world's geometric boundary.
4. The root's left child represents the world's left half. The node stores information about the left half's geometric boundary, and a list of all objects contained within.
5. The root's right child contains similar information for the right half.
6. Using the screen's position within the world as a lookup into the BSP tree quickly yields the right node's list of objects. A large number of objects are quickly eliminated from the list of potential objects on screen.
7. Further partitioning makes the tree even more useful.

PARTICIPATION ACTIVITY

7.2.5: Binary space partitioning.



- 1) When traversing down a BSP tree, half the objects are eliminated each level.

- True
- False



- 2) A BSP implementation could choose to split regions in arbitrary locations, instead of right down the middle.

- True
- False



- 3) In the animation, if the parts of the screen were in 2 different regions, then all objects from the 2 regions would have to be analyzed when rendering.

- True
- False

©zyBooks 06/06/23 23:54 169246
Taylor Larrechea
COLORADOCSPB2270Summer2023



- 4) BSP can be used in 3-D graphics as well as 2-D.

- True
- False

Using trees to store collections

©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Most of the tree data structures discussed in this book serve to store a collection of values. Numerous tree types exist to store data collections in a structured way that allows for fast searching, inserting, and removing of values.

7.3 Binary search



This section has been set as optional by your instructor.

Linear search vs. binary search

Linear search may require searching all list elements, which can lead to long runtimes. For example, searching for a contact on a smartphone one-by-one from first to last can be time consuming. Because a contact list is sorted, a faster search, known as binary search, checks the middle contact first. If the desired contact comes alphabetically before the middle contact, binary search will then search the first half and otherwise the last half. Each step reduces the contacts that need to be searched by half.

PARTICIPATION ACTIVITY

7.3.1: Using binary search to search contacts on your phone.



Animation captions:

©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

1. A contact list stores contacts sorted by name. Searching for Pooja using a binary search starts by checking the middle contact.
2. The middle contact is Muhammad. Pooja is alphabetically after Muhammad, so the binary search only searches the contacts after Muhammad. Only half the contacts now need to be searched.

3. Binary search continues by checking the middle element between Muhammad and the last contact. Pooja is before Sharod, so the search continues with only those contacts between Muhammad and Sharod, which is one fourth of the contacts.
4. The middle element between Muhammad and Sharod is Pooja. Each step reduces the number of contacts to search by half.

PARTICIPATION ACTIVITY

7.3.2: Using binary search to search a contact list.

©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



A contact list is searched for Bob.

Assume the following contact list: (Amy, Bob, Chris, Holly, Ray, Sarah, Zoe)

- 1) What is the first contact searched?

**Check****Show answer**

- 2) What is the second contact searched?

**Check****Show answer**

Binary search algorithm

Binary search is a faster algorithm for searching a list if the list's elements are sorted and directly accessible (such as an array). Binary search first checks the middle element of the list. If the search key is found, the algorithm returns the matching location. If the search key is not found, the algorithm repeats the search on the remaining left sublist (if the search key was less than the middle element) or the remaining right sublist (if the search key was greater than the middle element).

PARTICIPATION ACTIVITY

7.3.3: Binary search efficiently searches sorted list by reducing the search space by half each iteration.

©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



Animation captions:

1. Elements with indices between low and high remain to be searched.
2. Search starts by checking the middle element.
3. If search key is greater than element, then only elements in right sublist need to be searched.
4. Each iteration reduces search space by half. Search continues until key found or search space is empty.

Figure 7.3.1: Binary search algorithm.

©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

```
BinarySearch(numbers, numbersSize, key) {  
    mid = 0  
    low = 0  
    high = numbersSize - 1  
  
    while (high >= low) {  
        mid = (high + low) / 2  
        if (numbers[mid] < key) {  
            low = mid + 1  
        }  
        else if (numbers[mid] > key) {  
            high = mid - 1  
        }  
        else {  
            return mid  
        }  
    }  
  
    return -1 // not found  
}  
  
main() {  
    numbers = { 2, 4, 7, 10, 11, 32, 45, 87 }  
    NUMBERS_SIZE = 8  
    i = 0  
    key = 0  
    keyIndex = 0  
  
    print("NUMBERS: ")  
    for (i = 0; i < NUMBERS_SIZE; ++i) {  
        print(numbers[i] + " ")  
    }  
    printLine()  
  
    print("Enter a value: ")  
    key = getIntFromUser()  
  
    keyIndex = BinarySearch(numbers, NUMBERS_SIZE, key)  
  
    if (keyIndex == -1) {  
        printLine(key + " was not found.")  
    }  
    else {  
        printLine("Found " + key + " at index " + keyIndex + ".")  
    }  
}
```

```
NUMBERS: 2 4 7 10 11 32 45 87  
Enter a value: 10  
Found 10 at index 3.  
...  
NUMBERS: 2 4 7 10 11 32 45 87  
Enter a value: 17  
17 was not found.
```

©zyBooks 06/06/23 23:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

7.3.4: Binary search algorithm execution.



Given list: (4, 11, 17, 18, 25, 45, 63, 77, 89, 114).

- 1) How many list elements will be checked to find the value 77 using binary search?

Check**Show answer**

©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



- 2) How many list elements will be checked to find the value 17 using binary search?

Check**Show answer**

- 3) Given an array with 32 elements, how many list elements will be checked if the key is less than all elements in the list, using binary search?

Check**Show answer**

Binary search efficiency

Binary search is incredibly efficient in finding an element within a sorted list. During each iteration or step of the algorithm, binary search reduces the search space (i.e., the remaining elements to search within) by half. The search terminates when the element is found or the search space is empty (element not found). For a 32 element list, if the search key is not found, the search space is halved to have 16 elements, then 8, 4, 2, 1, and finally none, requiring only 6 steps. For an N element list, the maximum number of steps required to reduce the search space to an empty sublist is

. Ex:

PARTICIPATION ACTIVITY

7.3.5: Speed of linear search versus binary search to find a number within a sorted list.



Animation captions:

1. A binary search begins with the middle element of the list. Each subsequent search reduces the search space by half. Using binary search, a match was found with only 3 comparisons.
2. Using linear search, a match was found after 6 comparisons. Compared to a linear search, binary search is incredibly efficient in finding an element within a sorted list.

©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

If each comparison takes 1 μ s (1 microsecond), a binary search algorithm's runtime is at most 20 μ s to search a list with 1,000,000 elements, 21 μ s to search 2,000,000 elements, 22 μ s to search 4,000,000 elements, and so on. Ex: Searching Amazon's online store, which has more than 200 million items, requires less than 28 μ s; up to 7,000,000 times faster than linear search.

PARTICIPATION ACTIVITY

7.3.6: Linear and binary search efficiency.



- 1) Suppose a list of 1024 elements is searched with linear search. How many distinct list elements are compared against a search key that is less than all elements in the list?

elements

Check

[Show answer](#)



- 2) Suppose a sorted list of 1024 elements is searched with binary search. How many distinct list elements are compared against a search key that is less than all elements in the list?

elements

Check

[Show answer](#)



©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

CHALLENGE ACTIVITY

7.3.1: Binary search.



489394.3384924.qx3zqy7

Start

A last names list is searched for Boyd using binary search.

Last names list: (Boyd, Diaz, Ford, Hall, Hill, King, Lee, Long, Page, Webb)

What is the first last name searched?

Ex: Webb

©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

What is the second last name searched?

1

2

3

4

5

Check

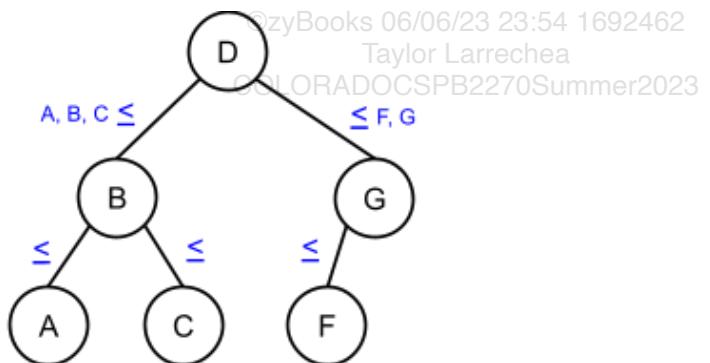
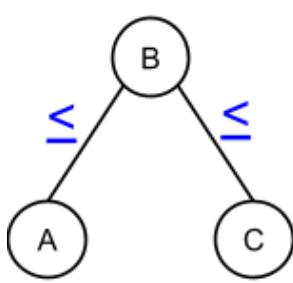
Next

7.4 Binary search trees

Binary search trees

An especially useful form of binary tree is a **binary search tree** (BST), which has an ordering property that any node's left subtree keys \leq the node's key, and the right subtree's keys \geq the node's key. That property enables fast searching for an item, as will be shown later.

Figure 7.4.1: BST ordering property: For three nodes, left child is less-than-or-equal-to parent, parent is less-than-or-equal-to right child. For more nodes, all keys in subtrees must satisfy the property, for every node.



PARTICIPATION ACTIVITY

7.4.1: BST ordering properties.

**Animation content:**

undefined

©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

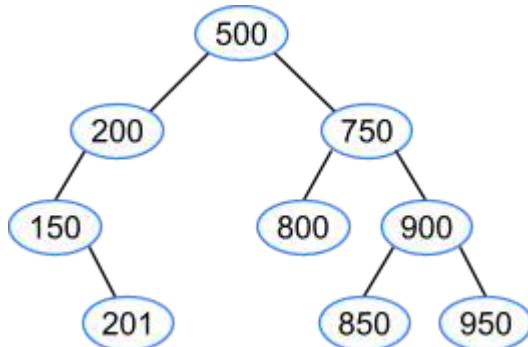
COLORADOCSPB2270Summer2023

Animation captions:

1. BST ordering property: Left subtree's keys \leq node's key, right subtree's keys \geq node's key.
2. All keys in subtree must obey the ordering property. Not a BST.
3. All keys in subtree must obey the ordering property. Not a BST.
4. All keys in subtree must obey the ordering property. Valid BST.

PARTICIPATION ACTIVITY

7.4.2: Binary search tree: Basic ordering property.



1) Does node 900 and the node's subtrees obey the BST ordering property?

- Yes
 No

2) Does node 750 and the node's subtrees obey the BST ordering property?

- Yes
 No

3) Does node 150 and the node's subtrees obey the BST ordering property?

- Yes
 No

©zyBooks 06/06/23 23:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



- 4) Does node 200 and the node's subtrees obey the BST ordering property?

Yes
 No

- 5) Is the tree a binary search tree?

Yes
 No

©zyBooks 06/06/23 23:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



- 6) Is the tree a binary tree?

Yes
 No



- 7) Would inserting 300 as the right child of 200 obey the BST ordering property (considering only nodes 300, 200, and 500)?

Yes
 No



Searching

To **search** nodes means to find a node with a desired key, if such a node exists. A BST may yield faster searches than a list. Searching a BST starts by visiting the root node (which is the first currentNode below):

Figure 7.4.2: Searching a BST.

```
if (currentNode->key == desiredKey) {  
    return currentNode; // The desired node was  
    found  
}  
else if (desiredKey < currentNode->key) {  
    // Visit left child, repeat  
}  
else if (desiredKey > currentNode->key) {  
    // Visit right child, repeat  
}
```

©zyBooks 06/06/23 23:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

If a child to be visited doesn't exist, the desired node does not exist. With this approach, only a small fraction of nodes need be compared.

PARTICIPATION ACTIVITY

7.4.3: A BST may yield faster searches than a list.


Animation captions:

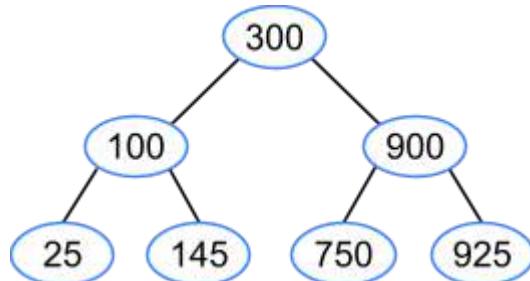
©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

1. Searching a 7-node list may require up to 7 comparisons. COLORADOCSPB2270Summer2023
2. In a BST, if desired key equals current node's key, return found. If less, descend to left child. If greater, descend to right child.
3. Searching a BST may require fewer comparisons, in this case 3 vs. 7.

PARTICIPATION ACTIVITY

7.4.4: Searching a BST.



- 1) In searching for 145, what node is visited first?



Check

[Show answer](#)

- 2) In searching for 145, what node is visited second?



Check

[Show answer](#)

- 3) In searching for 145, what node is visited third?



Check

[Show answer](#)

©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023





- 4) Which nodes would be visited when searching for 900? Write nodes in order visited, as: 5, 10

Check**Show answer**

- 5) Which nodes would be visited when searching for 800? Write nodes in order visited, as: 5, 10, 15

Check**Show answer**

- 6) What is the worst case (largest) number of nodes visited when searching for a key?

Check**Show answer**

©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



BST search runtime

Searching a BST in the worst case requires $H + 1$ comparisons, meaning $O(H)$ comparisons, where H is the tree height. Ex: A tree with a root node and one child has height 1; the worst case visits the root and the child: $1 + 1 = 2$. A major BST benefit is that an N -node binary tree's height may be as small as $O(\sqrt{N})$, yielding extremely fast searches. Ex: A 10,000 node list may require 10,000 comparisons, but a 10,000 node BST may require only 14 comparisons.

A binary tree's height can be minimized by keeping all levels full, except possibly the last level. Such an "all-but-last-level-full" binary tree's height is .

Table 7.4.1: Minimum binary tree heights for N nodes are equivalent to

©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Nodes N	Height H			Nodes per level
1	0	0	0	1

2	1	1	1	1/1
3	1	1.6	1	1/2
4	2	2	2	1/2/1
5	2	2.3	2	1/2/2
6	2	2.6	2	1/2/3
7	2	2.8	2	1/2/4
8	3	3	3	1/2/4/1
9	3	3.2	3	1/2/4/2
...				
15	3	3.9	3	1/2/4/8
16	4	4	4	1/2/4/8/1

©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea
COLORADO CSPB2270Summer2023**PARTICIPATION ACTIVITY**

7.4.5: Searching a perfect BST with N nodes requires only O() comparisons.

**Animation captions:**

1. A perfect binary tree has height .
2. A perfect binary tree search is O(), so O().
3. Searching a BST may be faster than searching a list.

PARTICIPATION ACTIVITY

7.4.6: Searching BSTs with N nodes.



What is the worst case (largest) number of comparisons given a BST with N nodes?

©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea
COLORADO CSPB2270Summer2023

- 1) Perfect BST with N = 7

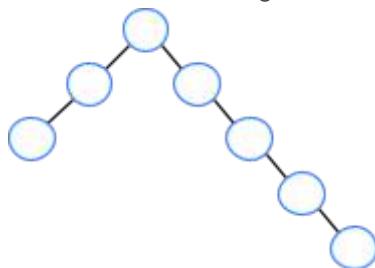
-
-
-

- 2) Perfect BST with N = 31



- 31
- 4
- 5

3) Given the following tree.



- 3
- 5

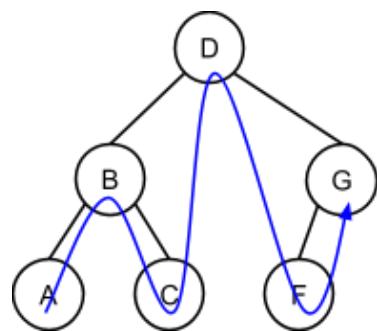
©zyBooks 06/06/23 23:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Successors and predecessors

A BST defines an ordering among nodes, from smallest to largest. A BST node's **successor** is the node that comes after in the BST ordering, so in A B C, A's successor is B, and B's successor is C. A BST node's **predecessor** is the node that comes before in the BST ordering.

If a node has a right subtree, the node's successor is that right subtree's leftmost child: Starting from the right subtree's root, follow left children until reaching a node with no left child (may be that subtree's root itself). If a node doesn't have a right subtree, the node's successor is the first ancestor having this node in a left subtree. Another section provides an algorithm for printing a BST's nodes in order.

Figure 7.4.3: A BST defines an ordering among nodes.



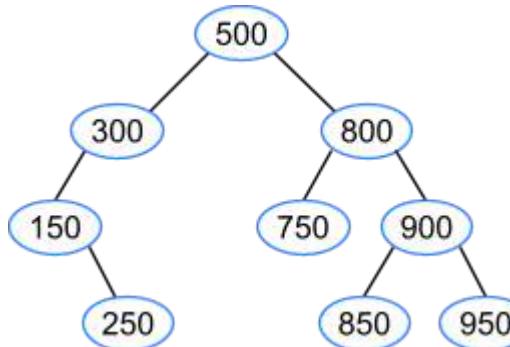
BST ordering:
A B C D F G

Successor follows in ordering.
Ex: D's successor is F.

©zyBooks 06/06/23 23:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

7.4.7: Binary search tree: Defined ordering.



©zyBooks 06/06/23 23:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 1) The first node in the BST ordering is 150.

True
 False

- 2) 150's successor is 250.

True
 False

- 3) 250's successor is 300.

True
 False

- 4) 500's successor is 850.

True
 False

- 5) 950's successor is 150.

True
 False

- 6) 950's predecessor is 900.

True
 False

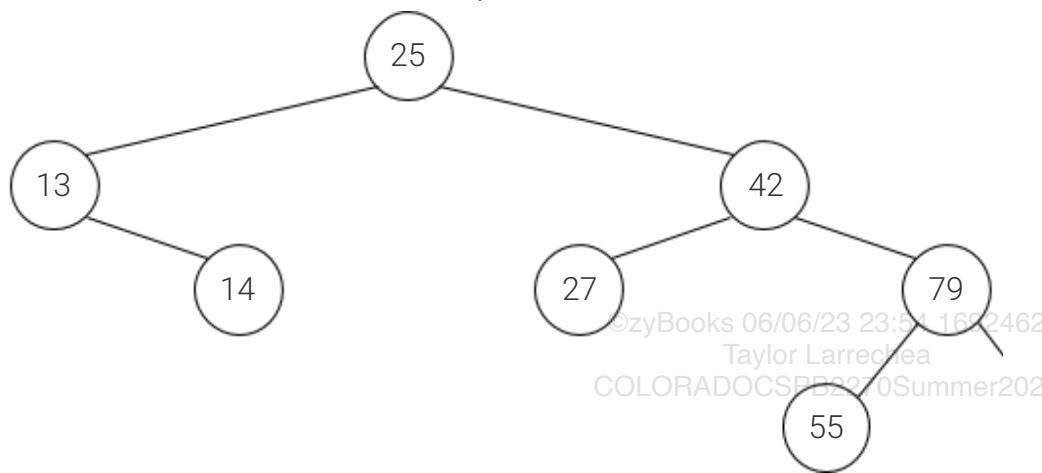
©zyBooks 06/06/23 23:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

CHALLENGE ACTIVITY

7.4.1: Binary search trees.

489394.3384924.qx3zqj7

Start



Does node 13 and the node's subtrees obey the BST ordering property? ▼

Does node 42 and the node's subtrees obey the BST ordering property? ▼

Is the tree a BST? ▼

1

2

3

4

5

Check

Next

7.5 BST search algorithm

Given a key, a **search** algorithm returns the first node found matching that key, or returns null if a matching node is not found. A simple BST search algorithm checks the current node (initially the tree's root), returning that node as a match, else assigning the current node with the left (if key is less) or right (if key is greater) child and repeating. If such a child is null, the algorithm returns null (matching node not found).

PARTICIPATION ACTIVITY

7.5.1: BST search algorithm.



©zyBooks 06/06/23 23:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Animation content:

undefined

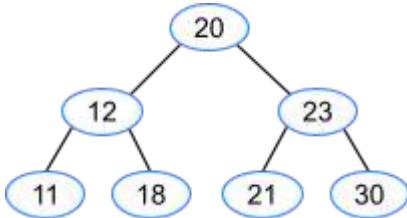
Animation captions:

1. BST search algorithm checks current node, returning a match if found. Otherwise, assigns current node with left (if key is less) or right (if key is greater) child and continues search.

2. If the child to be visited does not exist, the algorithm returns null indicating no match found.

PARTICIPATION ACTIVITY**7.5.2: BST search algorithm.**

Consider the following tree.



©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

1) When searching for key 21, what node is visited first?

- 20
- 21



2) When searching for key 21, what node is visited second?

- 12
- 23



3) When searching for key 21, what node is visited third?

- 21
- 30



4) If the current node matches the key, when does the algorithm return the node?

- Immediately
- Upon exiting the loop



5) If the child to be visited is null, when does the algorithm return null?

- Immediately
- Upon exiting the loop



6) What is the maximum loop iterations for a perfect binary tree with 7 nodes, if a node matches?



©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

3 7

- 7) What is the maximum loop iterations for a perfect binary tree with 7 nodes, if no node matches?

 3 7

©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

- 8) What is the maximum loop iterations for a perfect binary tree with 255 nodes?

 8 255

- 9) Suppose node 23 was instead 21, meaning two 21 nodes exist (which is allowed in a BST). When searching for 21, which node will be returned?

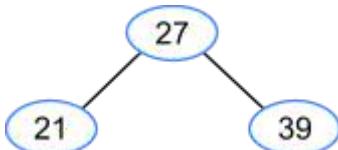
 Leaf Internal
PARTICIPATION ACTIVITY

7.5.3: BST search algorithm decisions.



Determine cur's next assignment given the key and current node.

- 1) key = 40, cur = 27

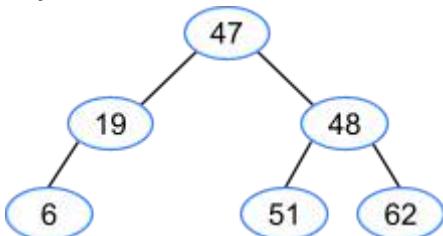
 27 21 39

©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

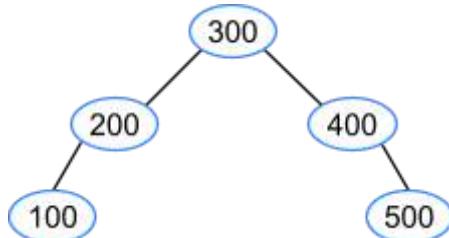
COLORADOCSPB2270Summer2023

- 2) key = 6, cur = 47



- 6
- 19
- 48

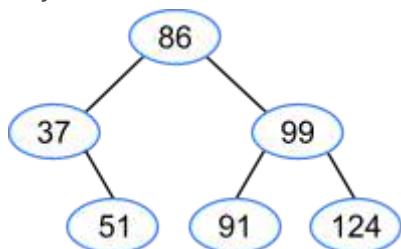
3) key 350, cur = 400



©zyBooks 06/06/23 23:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- Search terminates and returns null.
- 400
- 500

4) key 91, cur = 99



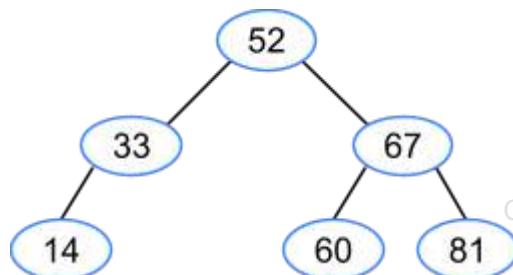
- 86
- 91
- 124

PARTICIPATION ACTIVITY

7.5.4: Tracing a BST search.



Consider the following tree. If node does not exist, enter null.



©zyBooks 06/06/23 23:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

1) When searching for key 45, what node is visited first?



//

Check**Show answer**

- 2) When searching for key 45, what node is visited second?

**Check****Show answer**

©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



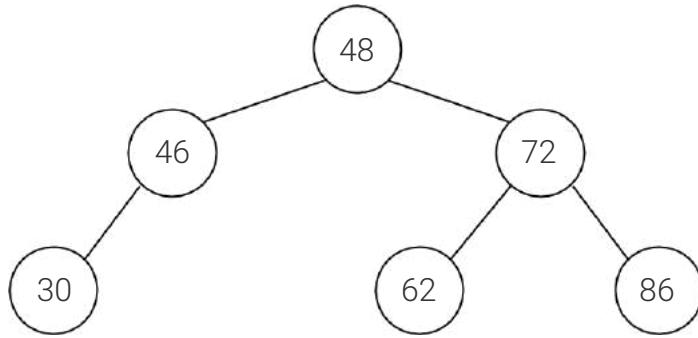
- 3) When searching for key 45, what node is visited third?

**Check****Show answer****CHALLENGE ACTIVITY**

7.5.1: BST search algorithm.



489394.3384924.qx3zqy7

Start

What does BSTSearch(tree, 50) return?



1

©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

?

Check**Next**

7.6 BST insert algorithm

Given a new node, a BST **insert** operation inserts the new node in a proper location obeying the BST ordering property. A simple BST insert algorithm compares the new node with the current node (initially the root).

- *Insert as left child:* If the new node's key is less than the current node, and the current node's left child is null, the algorithm assigns that node's left child with the new node.
- *Insert as right child:* If the new node's key is greater than or equal to the current node, and the current node's right child is null, the algorithm assigns the node's right child with the new node.
- *Search for insert location:* If the left (or right) child is not null, the algorithm assigns the current node with that child and continues searching for a proper insert location.

PARTICIPATION ACTIVITY

7.6.1: Binary search tree insertions.

**Animation content:**

undefined

Animation captions:

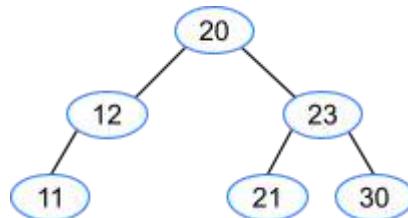
1. A node inserted into an empty tree will become the tree's root.
2. The BST is searched to find a suitable location to insert the new node as a leaf node.

PARTICIPATION ACTIVITY

7.6.2: BST insert algorithm.



Consider the following tree.



- 1) Where will a new node 18 be inserted?

- 12's right child
- 11's right child

- 2) Where will a new node 11 be inserted?

(So two nodes of 11 will exist).

- 11's left child

©zyBooks 06/06/23 23:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



11's right child

- 3) Assume a perfect 7-node BST. How many algorithm loop iterations will occur for an insert?

3

7

- 4) Assume a perfect 255-node BST. How many algorithm loop iterations will occur for an insert?

8

255

©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

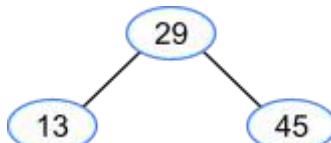
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

7.6.3: BST insert algorithm decisions.

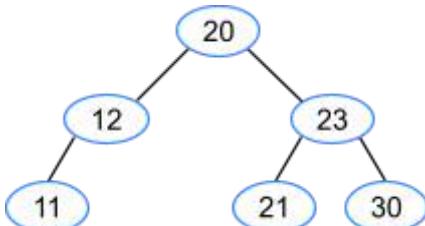
Determine the insertion algorithm's next step given the new node's key and the current node.

- 1) key = 7, currentNode = 29



- currentNode->left = node
- currentNode =
currentNode->right
- currentNode = currentNode->left

- 2) key = 18, currentNode = 12



- currentNode->left = node
- currentNode->right = node
- currentNode =
currentNode->right

- 3) key = 87, currentNode = null, tree->root
= null (empty tree)

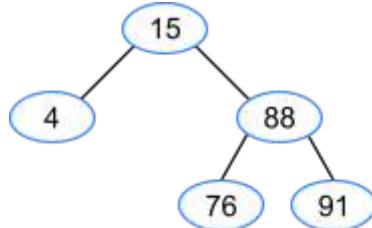
©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

- tree \rightarrow root = node
- currentNode \rightarrow right = node
- currentNode \rightarrow left = node

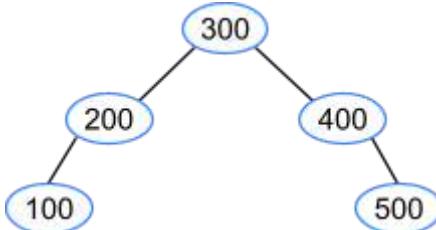
4) key = 53, currentNode = 76



©zyBooks 06/06/23 23:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- currentNode \rightarrow left = node
- currentNode \rightarrow right = node
- tree \rightarrow root = node

5) key = 600, currentNode = 400

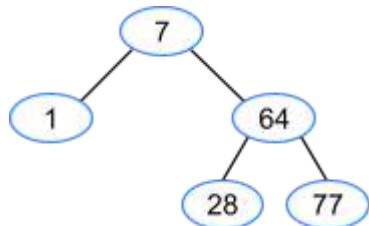


- currentNode \rightarrow left = node
- currentNode =
currentNode \rightarrow right
- currentNode \rightarrow right = node

**PARTICIPATION
ACTIVITY**

7.6.4: Tracing BST insertions.

Consider the following tree.



©zyBooks 06/06/23 23:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

1) When inserting a new node with key 35, what node is visited first?

Check

Show answer



- 2) When inserting a new node with key 35, what node is visited second?

 //**Check****Show answer**

- 3) When inserting a new node with key 35, what node is visited third?

 //**Check****Show answer**

- 4) Where is the new node inserted?

Type: left or right

 //**Check****Show answer**

BST insert algorithm complexity

The BST insert algorithm traverses the tree from the root to a leaf node to find the insertion location. One node is visited per level. A BST with N nodes has at least levels and at most levels. Therefore, the runtime complexity of insertion is best case and worst case .

The space complexity of insertion is because only a single pointer is used to traverse the tree to find the insertion location.

CHALLENGE ACTIVITY

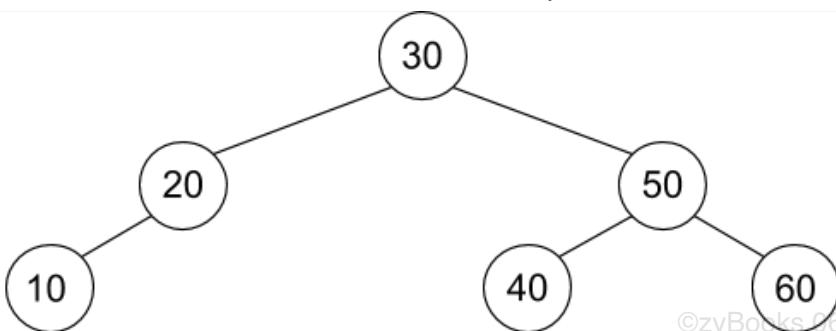
7.6.1: BST insert algorithm.

©zyBooks 06/06/23 23:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



489394.3384924.qx3zqy7

Start



©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Where will a new node 25 be inserted?

 child of node

1	2	3
---	---	---

[Check](#) [Next](#)

Exploring further:

- [Binary search tree visualization](#)

7.7 BST remove algorithm

Given a key, a BST **remove** operation removes the first-found matching node, restructuring the tree to preserve the BST ordering property. The algorithm first searches for a matching node just like the search algorithm. If found (call this node X), the algorithm performs one of the following sub-algorithms:

- *Remove a leaf node*: If X has a parent (so X is not the root), the parent's left or right child (whichever points to X) is assigned with null. Else, if X was the root, the root pointer is assigned with null, and the BST is now empty.
- *Remove an internal node with single child*: If X has a parent (so X is not the root), the parent's left or right child (whichever points to X) is assigned with X's single child. Else, if X was the root, the root pointer is assigned with X's single child.
- *Remove an internal node with two children*: This case is the hardest. First, the algorithm locates X's successor (the leftmost child of X's right subtree), and copies the successor to X. Then, the algorithm recursively removes the successor from the right subtree.

PARTICIPATION ACTIVITY

7.7.1: BST remove: Removing a leaf, or an internal node with a single child.

**Animation captions:**

1. Removing a leaf node: The parent's right child is assigned with null.
2. Remove an internal node with a single child: The parent's right child is assigned with node's single child.

©zyBooks 06/06/23 23:54 1692462
Taylor LarrecheaCOLORADO CSPB2270 Summer2023**PARTICIPATION ACTIVITY**

7.7.2: BST remove: Removing internal node with two children.

**Animation captions:**

1. Find successor: Leftmost child in node 25's right subtree is node 27.
2. Copy successor to current node.
3. Remove successor from right subtree.

Figure 7.7.1: BST remove algorithm.

©zyBooks 06/06/23 23:54 1692462
Taylor LarrecheaCOLORADO CSPB2270 Summer2023

```
BSTRemove(tree, key) {
    par = null
    cur = tree->root
    while (cur is not null) { // Search for node
        if (cur->key == key) { // Node found
            if (cur->left is null && cur->right is null) { // Remove leaf
                if (par is null) // Node is root
                    tree->root = null
                else if (par->left == cur)
                    par->left = null
                else
                    par->right = null
            }
            else if (cur->right is null) { // Remove node with
                only left child
                    if (par is null) // Node is root
                        tree->root = cur->left
                    else if (par->left == cur)
                        par->left = cur->left
                    else
                        par->right = cur->left
                }
            else if (cur->left is null) { // Remove node with
                only right child
                    if (par is null) // Node is root
                        tree->root = cur->right
                    else if (par->left == cur)
                        par->left = cur->right
                    else
                        par->right = cur->right
                }
            else { // Remove node with two
                children
                    // Find successor (leftmost child of right subtree)
                    suc = cur->right
                    while (suc->left is not null)
                        suc = suc->left
                    successorData = Create copy of suc's data
                    BSTRemove(tree, suc->key) // Remove successor
                    Assign cur's data with successorData
                }
            return // Node found and removed
        }
        else if (cur->key < key) { // Search right
            par = cur
            cur = cur->right
        }
        else { // Search left
            par = cur
            cur = cur->left
        }
    }
    return // Node not found
}
```

©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

BST remove algorithm complexity

The BST remove algorithm traverses the tree from the root to find the node to remove. When the node being removed has two children, the node's successor is found and a recursive call is made. One node is visited per level, and in the worst case scenario the tree is traversed twice from the root to a leaf. A BST with ~~nodes has at least~~ ©zyBooks 06/06/23 23:54 1692462 Taylor Larrechea COLORADOCSPB2270Summer2023 nodes has at least ~~levels and at most~~ ©zyBooks 06/06/23 23:54 1692462 Taylor Larrechea COLORADOCSPB2270Summer2023 levels and at most ~~levels~~ ©zyBooks 06/06/23 23:54 1692462 Taylor Larrechea COLORADOCSPB2270Summer2023 levels. So removal's worst case time complexity is ~~O(n^2)~~ ©zyBooks 06/06/23 23:54 1692462 Taylor Larrechea COLORADOCSPB2270Summer2023 for a BST with ~~levels~~ ©zyBooks 06/06/23 23:54 1692462 Taylor Larrechea COLORADOCSPB2270Summer2023 levels and worst case ~~O(n^2)~~ ©zyBooks 06/06/23 23:54 1692462 Taylor Larrechea COLORADOCSPB2270Summer2023 for a tree with ~~levels~~ ©zyBooks 06/06/23 23:54 1692462 Taylor Larrechea COLORADOCSPB2270Summer2023 levels.

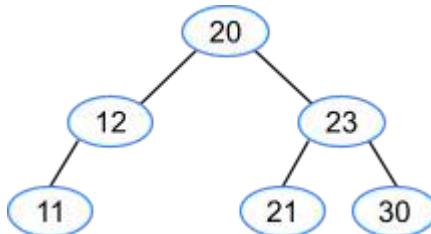
Two pointers are used to traverse the tree during removal. When the node being removed has two children, a third pointer and a copy of one node's data are also used, and one recursive call is made. So removal's space complexity is always ~~O(1)~~.

PARTICIPATION ACTIVITY

7.7.3: BST remove algorithm.



Consider the following tree. Each question starts from the original tree. Use this text notation for the tree: (20 (12 (11, -), 23 (21, 30))). The - means the child does not exist.



1) What is the tree after removing 21?



- (20 (12 (11, -), 23 (-, 30)))
- (20 (12 (11, -), 23))

2) What is the tree after removing 12?



- (20 (- (11, -), 23 (21, 30)))
- (20 (11, 23 (21, 30)))

3) What is the tree after removing 20?



- (21 (12 (11, -), 23 (-, 30)))
- (23 (12 (11, -), 30 (21, -)))

©zyBooks 06/06/23 23:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



- 4) Removing a node from an N-node nearly-full
BST has what computational complexity?

- O(\dots)
- O(\dots)

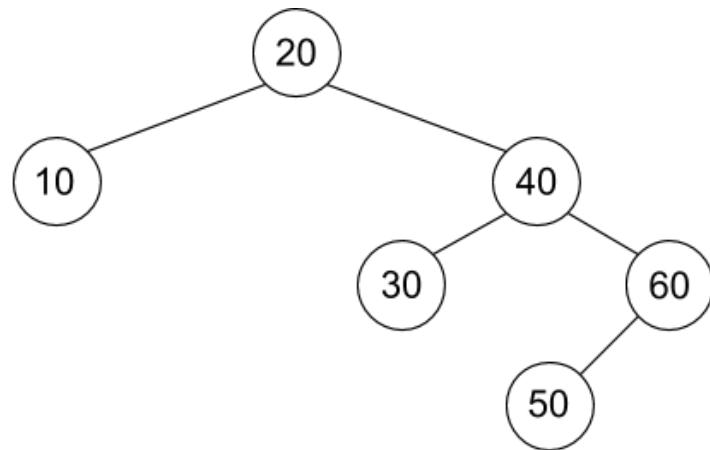
©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

CHALLENGE ACTIVITY**7.7.1: BST remove algorithm.**

489394.3384924.qx3zqy7

Start

BSTRemove(tree, 30) executes.

What is the left child of 40? What is the right child of 40?

1

2

3

Check**Next**

©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

7.8 BST inorder traversal

A **tree traversal** algorithm visits all nodes in the tree once and performs an operation on each node. An **inorder traversal** visits all nodes in a BST from smallest to largest, which is useful for example to print the tree's nodes in sorted order. Starting from the root, the algorithm recursively prints the left subtree, the current node, and the right subtree.

Figure 7.8.1: BST inorder traversal algorithm.

©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

```
BSTPrintInorder(node) {
    if (node is null)
        return

    BSTPrintInorder(node->left)
    Print node
    BSTPrintInorder(node->right)
}
```

PARTICIPATION ACTIVITY

7.8.1: BST inorder print algorithm.



Animation captions:

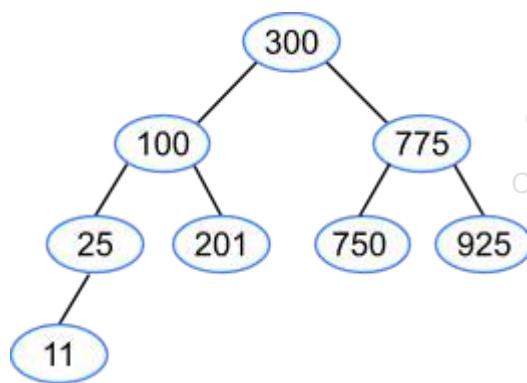
1. An inorder traversal starts at the root. Recursive call descends into left subtree.
2. When left done, current is printed, then recursively descend into right subtree.
3. Return from recursive call causes ascending back up the tree; left is done, so do current and right.
4. Continues similarly.

PARTICIPATION ACTIVITY

7.8.2: Inorder traversal of a BST.



Consider the following tree.



©zyBooks 06/06/23 23:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 1) What node is printed first?



//[Show answer](#)

- 2) Complete the tree traversal after node 300's left subtree has been printed.

11 25 100 201

 //[Show answer](#)

©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



- 3) How many nodes are visited?

 // //[Show answer](#)

- 4) Using left, current, and right, what ordering will print the BST from largest to smallest? Ex: An inorder traversal uses left current right.

 // //[Show answer](#)

7.9 BST height and insertion order

BST height and insertion order

Recall that a tree's **height** is the maximum edges from the root to any leaf. (Thus, a one-node tree has height 0.)

The *minimum* N-node binary tree height is _____, achieved when each level is full except possibly the last. The *maximum* N-node binary tree height is $N - 1$ (the $- 1$ is because the root is at height 0).

Searching a BST is fast if the tree's height is near the minimum. Inserting items in random order naturally keeps a BST's height near the minimum. In contrast, inserting items in nearly-sorted order leads to a nearly-maximum tree height.

**PARTICIPATION
ACTIVITY**

7.9.1: Inserting in random order keeps tree height near the minimum.
Inserting in sorted order yields the maximum.

**Animation captions:**

1. Inserting in random order naturally keeps tree height near the minimum, in this case 3
(minimum: 2)
2. Inserting in sorted order yields the maximum height, in this case 6.
3. If nodes are given beforehand, randomizing the ordering before inserting keeps tree height near minimum.

©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADOCSBP2270Summer2023

**PARTICIPATION
ACTIVITY**

7.9.2: BST height.



Draw a BST by hand, inserting nodes one at a time, to determine a BST's height.

- 1) A new BST is built by inserting nodes in this order:

6 2 8



What is the tree height?

(Remember, the root is at height 0)



Check

Show answer

- 2) A new BST is built by inserting nodes in this order:

20 12 23 18 30



What is the tree height?



Check

Show answer

©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADOCSBP2270Summer2023

- 3) A new BST is built by inserting nodes in this order:

30 23 21 20 18



What is the tree height?

Check**Show answer**

- 4) A new BST is built by inserting nodes in this order:

30 11 23 21 20

What is the tree height?

©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023


Check**Show answer**

- 5) A new BST is built by inserting 255 nodes in sorted order. What is the tree height?

Check**Show answer**

- 6) A new BST is built by inserting 255 nodes in random order. What is the minimum possible tree height?

Check**Show answer**

BSTGetHeight algorithm

Given a node representing a BST subtree, the height can be computed as follows:

- If the node is null, return -1.
- Otherwise recursively compute the left and right child subtree heights, and return 1 plus the greater of the 2 child subtrees' heights.

©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

PARTICIPATION
ACTIVITY

7.9.3: BSTGetHeight algorithm.



Animation content:

undefined

Animation captions:

1. BSTGetHeight(tree \rightarrow root) is called to get the height of the tree. The height of the root's left child is determined first using a recursive call.
2. BSTGetHeight for node 18 makes a recursive call on node 12. BSTGetHeight on node 12 makes a recursive call on the null left child, which returns -1. ©zyBooks 06/06/23 23:54 1692462 Taylor Larrechea
3. Returning to the BSTGetHeight(node 12) call, a recursive call is now made on the right child. Node 14 is a leaf, so both recursive calls return -1.
4. BSTGetHeight(node 14) returns $1 + \max(-1, -1) = 1 + -1 = 0$.
5. BSTGetHeight(node 12) has completed 2 recursive calls and returns $1 + \max(-1, 0) = 1$. BSTGetHeight(node 18) makes the recursive call on the null right child, which returns -1.
6. A recursive call is made for each node in the tree. BSTGetHeight(tree \rightarrow root) returns $1 + \max(2, 1) = 3$, which is the tree's height.

PARTICIPATION ACTIVITY

7.9.4: BSTGetHeight algorithm.



- 1) BSTGetHeight returns 0 for a tree with a single node.

- True
 False



- 2) The base case for BSTGetHeight is when the node argument is null.

- True
 False



- 3) The worst-case time complexity for BSTGetHeight is $O(\log N)$, where N is the number of nodes in the tree.

- True
 False



- 4) BSTGetHeight would also work if the recursive call on the right child was made before the recursive call on the left child.

- True
 False

©zyBooks 06/06/23 23:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



7.10 BST parent node pointers

A BST implementation often includes a parent pointer inside each node. A balanced BST, such as an AVL tree or red-black tree, may utilize the parent pointer to traverse up the tree from a particular node to find a node's parent, grandparent, or siblings. The BST insertion and removal algorithms below insert or remove nodes in a BST with nodes containing parent pointers.

©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADO CSPB2270Summer2023

Figure 7.10.1: BSTInsert algorithm for BSTs with nodes containing parent pointers.

```
BSTInsert(tree, node) {
    if (tree->root == null) {
        tree->root = node
        node->parent = null
        return
    }

    cur = tree->root
    while (cur != null) {
        if (node->key < cur->key) {
            if (cur->left == null)
            {
                cur->left = node
                node->parent = cur
                cur = null
            }
            else
                cur = cur->left
        }
        else {
            if (cur->right == null)
            {
                cur->right = node
                node->parent = cur
                cur = null
            }
            else
                cur = cur->right
        }
    }
}
```

©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADO CSPB2270Summer2023

Figure 7.10.2: BSTReplaceChild algorithm.

```
BSTReplaceChild(parent, currentChild,
newChild) {
    if (parent->left != currentChild &&
        parent->right != currentChild)
        return false

    if (parent->left == currentChild)
        parent->left = newChild
    else
        parent->right = newChild

    if (newChild != null)
        newChild->parent = parent
    return true
}
```

©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Figure 7.10.3: BSTRemoveKey and BSTRemoveNode algorithms for BSTs with nodes containing parent pointers.

©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

```
BSTRemoveKey(tree, key) {
    node = BSTSearch(tree, key)
    BSTRemoveNode(tree, node)
}

BSTRemoveNode(tree, node) {
    if (node == null)
        return

    // Case 1: Internal node with 2 children
    if (node->left != null && node->right != null) {
        // Find successor
        succNode = node->right
        while (succNode->left)
            succNode = succNode->left

        // Copy value/data from succNode to node
        node = Copy succNode

        // Recursively remove succNode
        BSTRemoveNode(tree, succNode)
    }

    // Case 2: Root node (with 1 or 0 children)
    else if (node == tree->root) {
        if (node->left != null)
            tree->root = node->left
        else
            tree->root = node->right

        // Make sure the new root, if non-null, has a null
parent
        if (tree->root != null)
            tree->root->parent = null
    }

    // Case 3: Internal with left child only
    else if (node->left != null)
        BSTReplaceChild(node->parent, node, node->left)

    // Case 4: Internal with right child only OR leaf
    else
        BSTReplaceChild(node->parent, node, node->right)
}
```

©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

7.10.1: BST parent node pointers.

©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

- 1) **BSTInsert** will not work if the tree's root is null.

 True

False

- 2) `BSTReplaceChild` will not work if the parent pointer is null.

 True False

- 3) `BSTRemoveKey` will not work if the key is not in the tree.

 True False

- 4) `BSTRemoveNode` will not work to remove the last node in a tree.

 True False

- 5) `BSTRemoveKey` uses `BSTRemoveNode`.

 True False

- 6) `BSTRemoveNode` uses `BSTRemoveKey`.

 True False

- 7) `BSTRemoveNode` may use recursion.

 True False

- 8) `BSTRemoveKey` will not properly update parent pointers when a non-root node is being removed.

 True False

- 9) All calls to `BSTRemoveNode` to remove a non-root node will result in a call to `BSTReplaceChild`.

 True

©zyBooks 06/06/23 23:54 169246
Taylor Larrechea
COLORADOCSPB2270Summer2023



©zyBooks 06/06/23 23:54 169246
Taylor Larrechea
COLORADOCSPB2270Summer2023



False

7.11 BST: Recursion

BST recursive search algorithm

©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

BST search can be implemented using recursion. A single node and search key are passed as arguments to the recursive search function. Two base cases exist. The first base case is when the node is null, in which case null is returned. If the node is non-null, then the search key is compared to the node's key. The second base case is when the search key equals the node's key, in which case the node is returned. If the search key is less than the node's key, a recursive call is made on the node's left child. If the search key is greater than the node's key, a recursive call is made on the node's right child.

PARTICIPATION ACTIVITY

7.11.1: BST recursive search algorithm.



Animation content:

undefined

Animation captions:

1. A call to BSTSearch(tree, 40) calls the BSTSearchRecursive function with the tree's root as the node argument.
2. The search key 40 is less than 64, so a recursive call is made on the root node's left child.
3. An additional recursive call searches node 32's right child. The key 40 is found and node 40 is returned.
4. Each function returns the result of a recursive call, so BSTSearch(tree, 40) returns node 40.

PARTICIPATION ACTIVITY

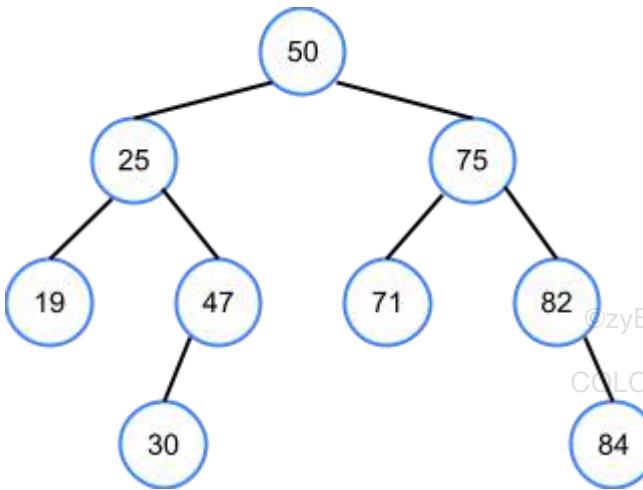
7.11.2: BST recursive search algorithm.



©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



©zyBooks 06/06/23 23:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 1) How many calls to BSTSearchRecursive are made by calling BSTSearch(tree, 71)? □

- 2
- 3
- 4

- 2) How many calls to BSTSearchRecursive are made by calling BSTSearch(tree, 49)? □

- 3
- 4
- 5

- 3) What is the maximum possible number of calls to BSTSearchRecursive when searching the tree? □

- 4
- 5

BST get parent algorithm

©zyBooks 06/06/23 23:54 1692462

A recursive BST get-parent algorithm searches for a parent in a way similar to the normal BST search algorithm. But instead of comparing the search key with a candidate node's key, the search key is compared with the keys of the candidate node's children.

Figure 7.11.1: BST get parent algorithm.

```
BSTGetParent(tree, node) {
    return BSTGetParentRecursive(tree->root, node)
}

BSTGetParentRecursive(subtreeRoot, node) {
    if (subtreeRoot is null)
        return null

    if (subtreeRoot->left == node or
        subtreeRoot->right == node) {
        return subtreeRoot
    }

    if (node->key < subtreeRoot->key) {
        return BSTGetParentRecursive(subtreeRoot->left,
node)
    }
    return BSTGetParentRecursive(subtreeRoot->right,
node)
}
```

©zyBooks 06/06/23 23:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY**7.11.3: BST get parent algorithm.**

- 1) BSTGetParent() returns null when the node parameter is the tree's root.

- True
- False

- 2) BSTGetParent() returns a leaf node when the node parameter is null.

- True
- False

- 3) The base case for BSTGetParentRecursive() is when subtreeRoot is null or is node's parent.

- True
- False

©zyBooks 06/06/23 23:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Recursive BST insertion and removal

BST insertion and removal can also be implemented using recursion. The insertion algorithm uses recursion to traverse down the tree until the insertion location is found. The removal algorithm uses

the recursive search functions to find the node and the node's parent, then removes the node from the tree. If the node to remove is an internal node with 2 children, the node's successor is recursively removed.

Figure 7.11.2: Recursive BST insertion and removal.

©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

©zyBooks 06/06/23 23:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

©zyBooks 06/06/23 23:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

```
BSTInsert(tree, node) {
    if (tree->root is null)
        tree->root = node
    else
        BSTInsertRecursive(tree->root, node)
}

BSTInsertRecursive(parent, nodeToInsert) {
    if (nodeToInsert->key < parent->key) { ©zyBooks 06/06/23 23:54 1692462
        if (parent->left is null)
            parent->left = nodeToInsert
        else
            BSTInsertRecursive(parent->left,
nodeToInsert)
    }
    else { Taylor Larrechea
        if (parent->right is null)
            parent->right = nodeToInsert
        else
            BSTInsertRecursive(parent->right,
nodeToInsert)
    }
}

BSTRemove(tree, key) {
    node = BSTSearch(tree, key)
    parent = BSTGetParent(tree, node)
    BSTRemoveNode(tree, parent, node)
}

BSTRemoveNode(tree, parent, node) {
    if (node == null)
        return false

    // Case 1: Internal node with 2 children
    if (node->left != null && node->right != null) { COLORADO CSPB2270 Summer 2023
        // Find successor and successor's parent
        succNode = node->right
        successorParent = node
        while (succNode->left != null) {
            successorParent = succNode
            succNode = succNode->left
        }

        // Copy the value from the successor node
        node = Copy succNode

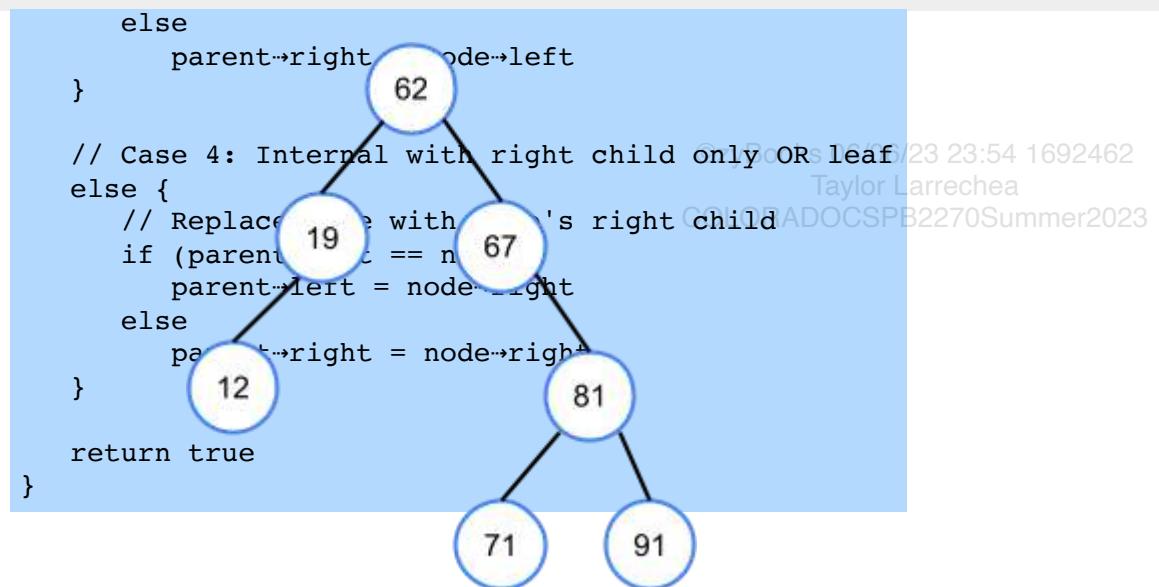
        // Recursively remove successor
        BSTRemoveNode(tree, successorParent, succNode) ©zyBooks 06/06/23 23:54 1692462
    }
}

// Case 2: Root node (with 1 or 0 children)
else if (node == tree->root) {
    if (node->left != null)
        tree->root = node->left
    else
        tree->root = node->right
}
```

```
// Case 3: Internal with left child only
else if (node->left != null) {
```

PARTICIPATION ACTIVITY

7.11.4: Recursive BST insertion and removal.



The following operations are executed on the above tree:

BSTInsert(tree, node 70)

BSTInsert(tree, node 56)

BSTRemove(tree, 67)

1) Where is node 70 inserted?

- Node 67's left child
- Node 71's left child
- Node 71's right child

2) How many times is BSTInsertRecursive called when inserting node 56?

- 2
- 3
- 5

3) How many times is BSTRemoveNode called when removing node 67?

- 1
- 2
- 3

4) What is the maximum number of calls to BSTRemoveNode when removing one of the tree's nodes?

- 2
- 4
- 5

7.12 Tries

©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Overview

A **trie** (or **prefix tree**) is a tree representing a set of strings. Each non-root node represents a single character. Each node has at most one child per distinct alphabet character. A **terminal node** is a node that represents a terminating character, which is the end of a string in the trie.

Tries provide efficient storage and quick search for strings, and are often used to implement auto-complete and predictive text input.

PARTICIPATION
ACTIVITY

7.12.1: Trie representing the set of strings: bat, cat, and cats.



Animation content:

undefined

Animation captions:

1. The following trie represents a set of two strings. Each string can be determined by traversing the path from the root to a leaf.
2. Suppose "cats" is added to the trie. Adding another child for 'c' would violate the trie requirements.
3. So the existing child for 'c' is reused.
4. Similarly, nodes for 'a' and 't' are reused. Node 't' has a new child added for 's'.
5. Exactly one terminal node exists for each string. Other nodes may be shared by multiple strings.

©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

PARTICIPATION
ACTIVITY

7.12.2: Trie representing the set of strings: bat, cat, and cats.



Refer to the trie above.

- 1) The terminal nodes can be removed, and instead the last character of a



string can be a leaf.

- True
- False

2) Adding the string "balance" would create a new branch off the root node.

- True
- False

©zyBooks 06/06/23 23:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

3) Inserting a string that doesn't already exist in the trie requires allocation of at least 1 new node.

- True
- False



Trie insert algorithm

Given a string, a **trie insert** operation creates a path from the root to a terminal node that visits all the string's characters in sequence.

A current node pointer initially points to the root. A loop then iterates through the string's characters. For each character C:

1. A new child node is added only if the current node does not have a child for C.
2. The current node pointer is assigned with the current node's child for C.

After all characters are processed, a terminal node is added and returned.

PARTICIPATION ACTIVITY

7.12.3: Trie insert algorithm.



Animation content:

undefined

Animation captions:

©zyBooks 06/06/23 23:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

1. The trie starts with an empty root node. Adding "APPLE" first adds a new child for 'A' to the root node.
2. New nodes are built for each remaining character in "APPLE".
3. The terminal node is added, completing insertion of "APPLE".
4. When adding "APPLY", the first 4 character nodes are reused.
5. Node L has no child for 'Y', so a new node is added. The terminal node is also added.

6. When adding "APP", only 1 new node is needed, the terminal node.

PARTICIPATION ACTIVITY**7.12.4: Trie insert algorithm.**

Assume a trie is built by executing the following code.

```
trieRoot = new TrieNode()
TrieInsert(trieRoot, "cat")
TrieInsert(trieRoot, "cow")
TrieInsert(trieRoot, "crow")
```

©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

- 1) When inserting "cat", ____ new nodes are created.

- 1
- 3
- 4



- 2) When inserting "crow", ____ new nodes are created.

- 1
- 4
- 5



- 3) If `TrieInsert(trieRoot, "cow")` is called a second time, ____ new nodes are created.

- 0
- 1
- 4



Trie search algorithm

Given a string, a **trie search** operation returns the terminal node corresponding to that string, or null if the string is not in the trie.

©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY**7.12.5: Trie search algorithm.**

Animation content:

undefined

Animation captions:

1. The search for "PAPAYA" starts at the root and iterates through one node per character. The terminal node is returned, indicating that the string was found.
2. The search for "GRAPE" ends quickly, since the root has no child for 'G'.
3. The search for "PEA" gets to the node for 'A'. No terminal child node exists after 'A', so null is returned.

©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

7.12.6: Trie search algorithm.



Refer to the trie above. Assume that to "visit" a node means accessing the node's children in TrieSearch.

- 1) `TrieSearch(root, "PINEAPPLE")` returns ____.

- the trie's root node
- the terminal node for "APPLE"
- null



- 2) When searching for "PLUM", ____ visited.

- only the root node is
- the root and the root's 'P' child node are
- all nodes in the root's 'P' child subtree are



- 3) `TrieSearch` will visit at most ____ nodes in a single search.

- 7
- 9
- 41



©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Trie remove algorithm

Given a string, a **trie remove** operation removes the string's corresponding terminal node and all non-root ancestors with 0 children.

PARTICIPATION ACTIVITY

7.12.7: Trie remove algorithm.



Animation content:

undefined

Animation captions:

1. TrieRemove is called to remove "BANANA". TrieRemove then calls TrieRemoveRecursive, passing the trie's root, the string "BANANA", and a character index of 0.
@zyBooks 06/06/23 23:54 1692462
Taylor Larrechea
2. The root has a child for 'B'. A recursive removal call is made for the child and the next character index.
#other2023
3. Recursive calls continue until the terminal node's parent is reached.
4. The terminal node is removed from the node's children and true is returned.
5. After returning from each recursive call, child nodes with 0 children are also removed.
6. When removing "APPLE", 4 nodes are removed.
7. Removal operations only remove nodes that are exclusive to the string being removed.

PARTICIPATION ACTIVITY

7.12.8: Trie remove algorithm.



Refer to the trie above. Match each operation to the statement that is true when the operation executes. Assume that "BANANA" and "APPLE" have already been removed.

If unable to drag and drop, refresh the page.

`TrieRemove(root, "AVOCADO")
TrieRemove(root, "APRICOT")`

`TrieRemove(root, "PAPAYA")`

`TrieRemove(root, "PEAR")`

`TrieRemove(root, "CHERRY")
TrieRemove(root, "PLUM")`

Remove's the root's child node for 'A'.

charIndex is 6 at the moment a terminal node is removed.

Has no effect.

Removes a total of 2 nodes from the trie.

Reset

Trie time complexities

Implementations commonly use a lookup table for a trie node's children, allowing retrieval of a child node from a character in $O(1)$ time. Therefore, to insert, remove, or search for a string of length M in a trie takes $O(M)$ time. The trie's current size does not affect each operation's time complexity.

©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

CHALLENGE ACTIVITY

7.12.1: Tries.



489394.3384924.qx3zqy7

Start

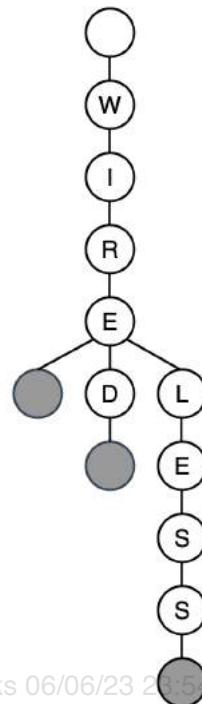
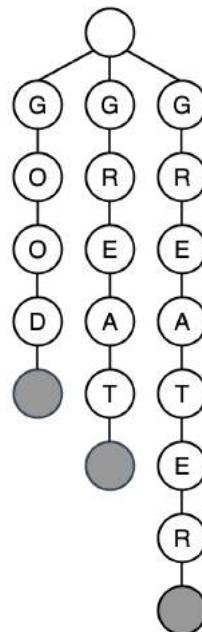
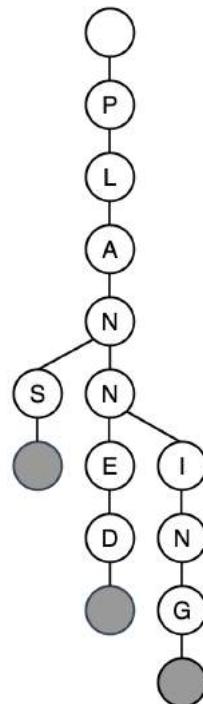
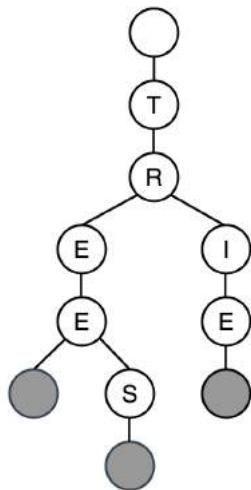
Select all valid tries.

A

B

C

D



©zyBooks 06/06/23 23:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

1

2

3

4

Check

Next

8.1 Red-black tree: A balanced tree

A **red-black tree** is a BST with two node types, namely red and black, and supporting operations that ensure the tree is balanced when a node is inserted or removed. The below red-black tree's requirements ensure that a tree with N nodes will have a height of O(log N).

©zyBooks 06/15/23 12:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

- Every node is colored either red or black.
- The root node is black.
- A red node's children cannot be red.
- A null child is considered to be a black leaf node.
- All paths from a node to any null leaf descendant node must have the same number of black nodes.

PARTICIPATION ACTIVITY

8.1.1: Red-black tree rules.



Animation captions:

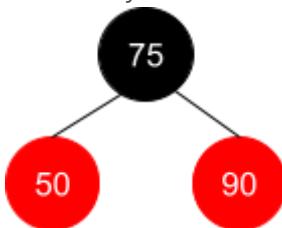
1. The null child pointer of a leaf node is considered a null leaf node and is always black. Visualizing null leaf nodes helps determine if a tree is a valid red-black tree.
2. Each requirement must be met for the tree to be a valid red-black tree.
3. A tree that violates any requirement is not a valid red-black tree.

PARTICIPATION ACTIVITY

8.1.2: Red-black tree rules.



- 1) Which red-black tree requirement does this BST not satisfy?



- Root node must be black.
- A red node's children cannot be red.
- All paths from a node to null leaf nodes must have the same number of black nodes.
- None.

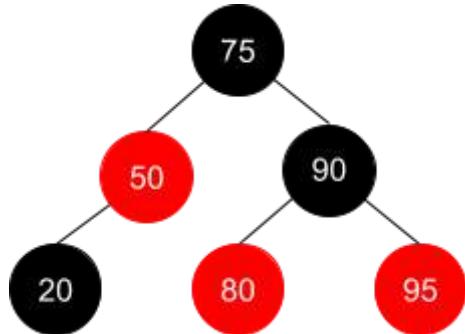
©zyBooks 06/15/23 12:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



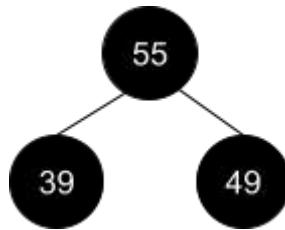
- 2) Which red-black tree requirement does this BST not satisfy?



©zyBooks 06/15/23 12:54 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

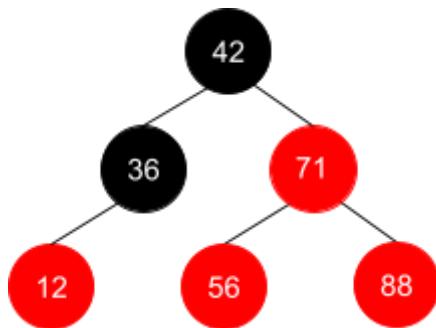
- Root node must be black.
- A red node's children cannot be red.
- Not all levels are full.
- All paths from a node to null leaf nodes must have the same number of black nodes.

- 3) The tree below is a valid red-black tree.



- True
- False

- 4) What single color change will make the below tree a valid red-black tree?



©zyBooks 06/15/23 12:54 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

- Change node 36's color to red.
- Change node 71's color to black.
- Change node 88's color to black.
- No single color change will make this a valid red-black tree..



- 5) A black node's children will always be the same color.

True
 False

- 6) All valid red-black trees will have more red nodes than black nodes.

True
 False

- 7) Any BST can be made a red-black tree by coloring all nodes black.

True
 False

©zyBooks 06/15/23 12:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

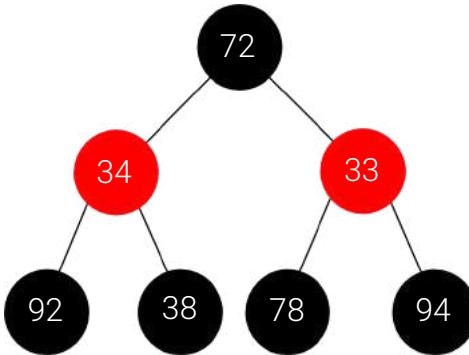
CHALLENGE ACTIVITY

8.1.1: Red-black tree: A balanced tree.



489394.3384924.qx3zqy7

Start



Is this binary tree a valid red-black tree? Select all that apply.

- Yes
 No, root node must be black
 No, a red node's child cannot be red
 No, all paths from a node to null leaf nodes must have the same number of black nodes
 No, BST ordering property violated

©zyBooks 06/15/23 12:54 1692462

COLORADOCSPB2270Summer2023

1

2

3

Check

Next

8.2 Red-black tree: Rotations

Introduction to rotations

A rotation is a local rearrangement of a BST that maintains the BST ordering property while rebalancing the tree. Rotations are used during the insert and remove operations on a red-black tree to ensure that red-black tree requirements hold. Rotating is said to be done "at" a node. A left rotation at a node causes the node's right child to take the node's place in the tree. A right rotation at a node causes the node's left child to take the node's place in the tree.

PARTICIPATION ACTIVITY

8.2.1: A simple left rotation in a red-black tree.

**Animation captions:**

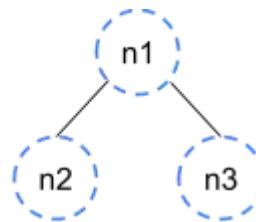
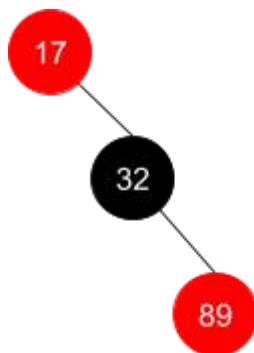
1. This BST is not a valid red-black tree. From the root, paths down to null leaves are inconsistent in terms of number of black nodes.
2. A left rotation at node 16 creates a valid red-black tree.

PARTICIPATION ACTIVITY

8.2.2: Red-black tree rotate left: 3 nodes.



Rotate left at node 17. Match the node value to the corresponding location in the rotated red-black tree template on the right.



If unable to drag and drop, refresh the page.

©zyBooks 06/15/23 12:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

17 89 32

n2

n1

n3

Reset

Left rotation algorithm

©zyBooks 06/15/23 12:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

A rotation requires altering up to 3 child subtree pointers. A left rotation at a node requires the node's right child to be non-null. Two utility functions are used for red-black tree rotations. The `RBTreeSetChild` utility function sets a node's left child, if the `whichChild` parameter is "left", or right child, if the `whichChild` parameter is "right", and updates the child's parent pointer. The `RBTreeReplaceChild` utility function replaces a node's left or right child pointer with a new value.

Figure 8.2.1: `RBTreeSetChild` utility function.

```
RBTreeSetChild(parent, whichChild, child) {
    if (whichChild != "left" && whichChild != "right")
        return false

    if (whichChild == "left")
        parent->left = child
    else
        parent->right = child
    if (child != null)
        child->parent = parent
    return true
}
```

Figure 8.2.2: `RBTreeReplaceChild` utility function.

```
RBTreeReplaceChild(parent, currentChild, newChild)
{
    if (parent->left == currentChild)
        return RBTreeSetChild(parent, "left",
newChild)
    else if (parent->right == currentChild)
        return RBTreeSetChild(parent, "right",
newChild)
    return false
}
```

The `RBTreeRotateLeft` function performs a left rotation at the specified node by updating the right child's left child to point to the node, and updating the node's right child to point to the right child's former left child. If non-null, the node's parent has the child pointer changed from node to the node's right child. Otherwise, if the node's parent is null, then the tree's root pointer is updated to point to the node's right child.

Figure 8.2.3: RBTreeRotateLeft pseudocode.

©zyBooks 06/15/23 12:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

```
RBTreeRotateLeft(tree, node) {  
    rightLeftChild = node->right->left  
    if (node->parent != null)  
        RBTreeReplaceChild(node->parent, node,  
        node->right)  
    else { // node is root  
        tree->root = node->right  
        tree->root->parent = null  
    }  
    RBTreeSetChild(node->right, "left", node)  
    RBTreeSetChild(node, "right", rightLeftChild)  
}
```

PARTICIPATION
ACTIVITY

8.2.3: RBTreeRotateLeft algorithm.



If unable to drag and drop, refresh the page.

Node with null left child

Red node

Node with null right child

Root node

RBTreeRotateLeft will not work when called at this type of node.

RBTreeRotateLeft called at this node requires the tree's root pointer to be updated.

©zyBooks 06/15/23 12:54 1692462

Taylor Larrechea

After calling RBTreeRotateLeft at this node, the node will have a null left child.

After calling RBTreeRotateLeft at this node, the node will be colored red.

Reset

Right rotation algorithm

Right rotation is analogous to left rotation. A right rotation at a node requires the node's left child to be non-null.

©zyBooks 06/15/23 12:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

8.2.4: RBTreeRotateRight algorithm.



Animation content:

undefined

Animation captions:

1. A right rotation at node 80 causes node 61 to become the new root, and nodes 40 and 80 to become the root's left and right children, respectively.
2. The rotation results in a valid red-black tree.

PARTICIPATION ACTIVITY

8.2.5: Right rotation algorithm.



- 1) A rotation will never change the root node's value.
 - True
 - False
- 2) A rotation at a node will only change properties of the node's descendants, but will never change properties of the node's ancestors.
 - True
 - False
- 3) RBTreeRotateRight works even if the node's parent is null.
 - True
 - False

©zyBooks 06/15/23 12:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023





- 4) RBTreeRotateRight works even if the node's left child is null.

- True
- False

- 5) RBTreeRotateRight works even if the node's right child is null.

- True
- False

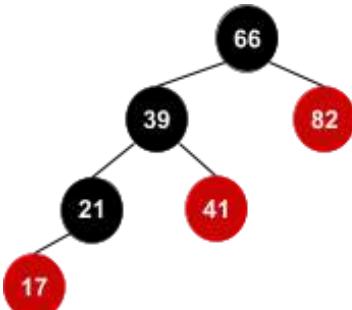
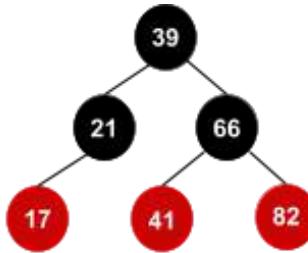
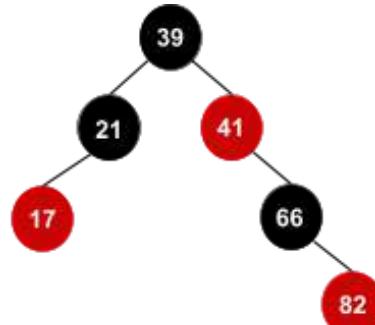
©zyBooks 06/15/23 12:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY
8.2.6: Red-black tree rotations.


Consider the three trees below:

**Tree 1****Tree 2****Tree 3**

- 1) Which trees are valid red-black trees?

- Tree 1 only
- Tree 2 only
- Tree 3 only
- All are valid red-black trees

- 2) Which operation on tree 1 would produce tree 2?

- Rotate right at node 82
- Rotate left at node 66
- Rotate right at node 66
- Rotate left at node 39

- 3) Which operation on tree 3 yields tree 2?

- Rotate left at node 21

©zyBooks 06/15/23 12:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

- Rotate left at node 39
 - Rotate left at node 41
 - Rotate left at node 66
- 4) A right rotation at node 21 in tree 2 yields a valid red-black tree.
- True
 - False

©zyBooks 06/15/23 12:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

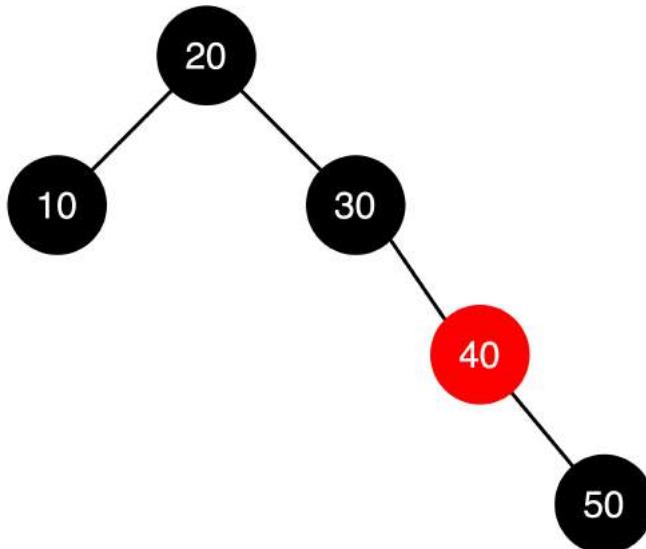
**CHALLENGE
ACTIVITY**

8.2.1: Red-black tree: Rotations.

489394.3384924.qx3zqy7

Start

An invalid red-black tree is shown below.



A ✓ rotation at node ✓ yields a valid red-black tree.

©zyBooks 06/15/23 12:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

1

2

3

Check

Next

8.3 Red-black tree: Insertion

Given a new node, a red-black tree **insert** operation inserts the new node in the proper location such that all red-black tree requirements still hold after the insertion completes.

Red-black tree insertion begins by calling **BSTInsert** to insert the node using the BST insertion rules. The newly inserted node is colored red and then a balance operation is performed on this node.⁴⁶²

Taylor Larrechea

COLORADO CSPB2270 Summer 2023

Figure 8.3.1: RBTreeInsert algorithm.

```
RBTreeInsert(tree, node) {
    BSTInsert(tree, node)
    node->color = red
    RBTreeBalance(tree,
    node)
}
```

The red-black balance operation consists of the steps below.

1. Assign **parent** with **node**'s parent, **uncle** with **node**'s uncle, which is a sibling of **parent**, and **grandparent** with **node**'s grandparent.
2. If **node** is the tree's root, then color **node** black and return.
3. If **parent** is black, then return without any alterations.
4. If **parent** and **uncle** are both red, then color **parent** and **uncle** black, color **grandparent** red, recursively balance **grandparent**, then return.
5. If **node** is **parent**'s right child and **parent** is **grandparent**'s left child, then rotate left at **parent**, assign **node** with **parent**, assign **parent** with **node**'s parent, and go to step 7.
6. If **node** is **parent**'s left child and **parent** is **grandparent**'s right child, then rotate right at **parent**, assign **node** with **parent**, assign **parent** with **node**'s parent, and go to step 7.
7. Color **parent** black and **grandparent** red.
8. If **node** is **parent**'s left child, then rotate right at **grandparent**, otherwise rotate left at **grandparent**.

The RBTreeBalance function uses the RBTreeGetGrandparent and RBTreeGetUncle utility functions to determine a node's grandparent and uncle, respectively.

©zyBooks 06/15/23 12:54 1692462

Taylor Larrechea

COLORADO CSPB2270 Summer 2023

Figure 8.3.2: RBTreeGetGrandparent and RBTreeGetUncle utility functions.

```
RBTreeGetGrandparent(node) {  
    if (node->parent == null)  
        return null  
    return node->parent->parent  
}  
  
RBTreeGetUncle(node) {  
    grandparent = null  
    if (node->parent != null)  
        grandparent =  
node->parent->parent  
        if (grandparent == null)  
            return null  
        if (grandparent->left ==  
node->parent)  
            return grandparent->right  
        else  
            return grandparent->left  
}
```

©zyBooks 06/15/23 12:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

8.3.1: RBTreeBalance algorithm.

**Animation content:**

undefined

Animation captions:

1. Insertion of 22 as the root starts with the normal BST insertion, followed by coloring the node red. The balance operation simply changes the root node to black.
2. Insertion of 11 and 33 do not require any node color changes or rotations.
3. Insertion of 55 requires recoloring the parent, uncle, and grandparent, then recursively balancing the grandparent.
4. Inserting 44 requires two rotations. The first rotation is a right rotation at the parent, node 55. The second rotation is a left rotation at the grandparent, node 33.

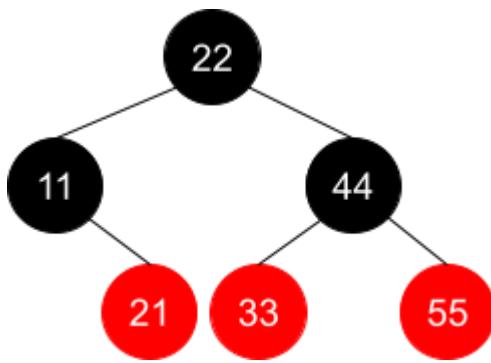
PARTICIPATION ACTIVITY

8.3.2: Red-black tree: insertion.

©zyBooks 06/15/23 12:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



Consider the following tree:



©zyBooks 06/15/23 12:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

- 1) Starting at and including the root node, how many black nodes are encountered on any path down to and including the null leaf nodes?

- 1
- 2
- 3
- 4

- 2) Insertion of which value will require at least 1 rotation?

- 10
- 20
- 30
- 45

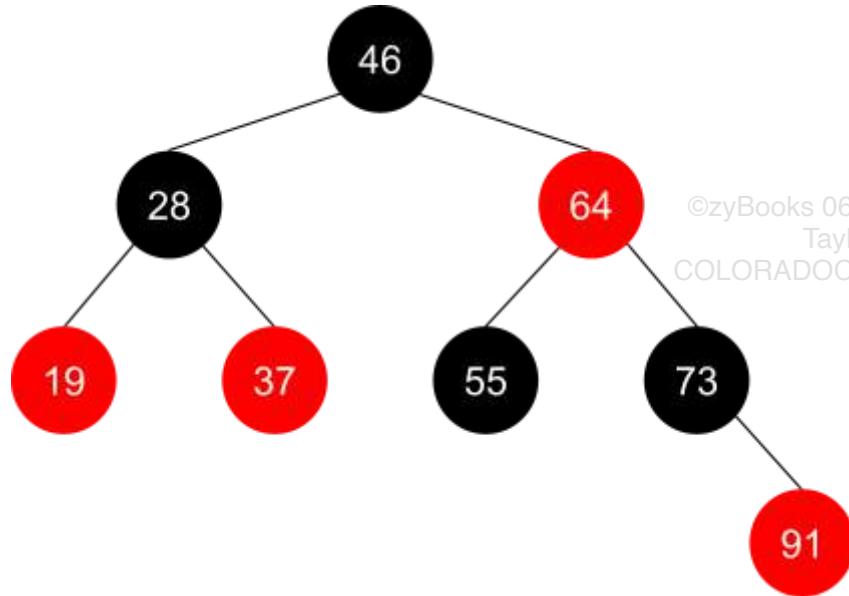
- 3) The values 11, 21, 22, 33, 44, 55 can be inserted in any order and the above tree will always be the result.

- True
- False

- 4) All red nodes could be recolored to black and the above tree would still be a valid red-black tree.

- True
- False

Select the order of tree-altering operations that occur as a result of calling RBTreeInsert to insert 82 into this tree:



If unable to drag and drop, refresh the page.

2 **4** **5** **1** **3** **Never**

Rotate left at node 73.

Insert red node 82 as node 91's left child.

Color grandparent node red.

Color parent node black.

Rotate right at node 91.

Call RBTreeBalance recursively on node 73.

©zyBooks 06/15/23 12:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

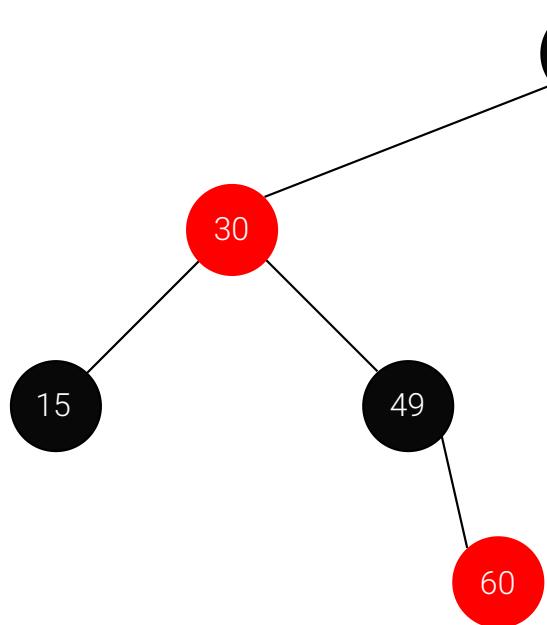
Reset

CHALLENGE ACTIVITY

8.3.1: Red-black tree: Insertion.



489394.3384924.qx3zqy7

Start

©zyBooks 06/15/23 12:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Node 11 is inserted. Prior to any rotations, what is node 11's _____?

Parent node: Ex: 10 or null

Grandparent node:

Uncle node:

Is a rotation required to complete the insertion?

1

2

3

Check**Next**

8.4 Red-black tree: Removal

Removal overview

Given a key, a red-black tree **remove** operation removes the first-found matching node, restructuring the tree to preserve all red-black tree requirements. First, the node to remove is found using **BSTSearch**. If the node is found, **RBTTreeRemoveNode** is called to remove the node.

Figure 8.4.1: RBTreeRemove algorithm.

```
RBTreeRemove(tree, key) {
    node = BSTSearch(tree, key)
    if (node != null)
        RBTreeRemoveNode(tree,
node)
}
```

©zyBooks 06/15/23 12:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

The RBTreeRemove algorithm consists of the following steps:

1. If the node has 2 children, copy the node's predecessor to a temporary value, recursively remove the predecessor from the tree, replace the node's key with the temporary value, and return.
2. If the node is black, call `RBTreePrepareForRemoval` to restructure the tree in preparation for the node's removal.
3. Remove the node using the standard BST `BSTRemove` algorithm.

Figure 8.4.2: RBTreeRemoveNode algorithm.

```
RBTreeRemoveNode(tree, node) {
    if (node->left != null && node->right != null) {
        predecessorNode = RBTreeGetPredecessor(node)
        predecessorKey = predecessorNode->key
        RBTreeRemoveNode(tree, predecessorNode)
        node->key = predecessorKey
        return
    }

    if (node->color == black)
        RBTreePrepareForRemoval(node)
        BSTRemove(tree, node->key)
}
```

Figure 8.4.3: RBTreeGetPredecessor utility function.

```
RBTreeGetPredecessor(node) {
    node = node->left
    while (node->right != null)
    {
        node = node->right
    }
    return node
}
```

©zyBooks 06/15/23 12:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

**PARTICIPATION
ACTIVITY**

8.4.1: Removal concepts.



1) The red-black tree removal algorithm uses the normal BST removal algorithm.

- True
- False

©zyBooks 06/15/23 12:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



2) RBTreeRemove uses the BST search algorithm.

- True
- False



3) Removing a red node with RBTreeRemoveNode will never cause RBTreePrepareForRemoval to be called.

- True
- False



4) Although RBTreeRemoveNode uses the node's predecessor, the algorithm could also use the successor.

- True
- False



Removal utility functions

Utility functions help simplify red-black tree removal code. The **RBTreeGetSibling** function returns the sibling of a node. The **RBTreeIsNotNullAndRed** function returns true only if a node is non-null and red, false otherwise. The **RBTreeIsNullOrBlack** function returns true if a node is null or black, false otherwise. The **RBTreeAreBothChildrenBlack** function returns true only if both of a node's children are black. Each utility function considers a null node to be a black node.

©zyBooks 06/15/23 12:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Figure 8.4.4: RBTreeGetSibling algorithm.

```
RBTreeGetSibling(node) {  
    if (node->parent != null) {  
        if (node ==  
            node->parent->left)  
            return node->parent->right  
        return node->parent->left  
    }  
    return null  
}
```

©zyBooks 06/15/23 12:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Figure 8.4.5: RBTreeIsNotNullAndRed algorithm.

```
RBTreeIsNotNullAndRed(node)  
{  
    if (node == null)  
        return false  
    return (node->color ==  
red)  
}
```

Figure 8.4.6: RBTreeIsNullOrBlack algorithm.

```
RBTreeIsNullOrBlack(node) {  
    if (node == null)  
        return true  
    return (node->color ==  
black)  
}
```

Figure 8.4.7: RBTreeAreBothChildrenBlack algorithm.

```
RBTreeAreBothChildrenBlack(node) {  
    if (node->left != null && node->left->color ==  
red)  
        return false  
    if (node->right != null && node->right->color ==  
red)  
        return false  
    return true  
}
```

©zyBooks 06/15/23 12:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

8.4.2: Removal utility functions.



- 1) Under what circumstance will RBTreeAreBothChildrenBlack always return true?
- When both of the node's children are null
 - When both of the node's children are non-null
 - When the node's left child is null
 - When the node's right child is null

©zyBooks 06/15/23 12:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



- 2) RBTreesNotNullAndRed will not work properly when passed a null node.

- True
- False



- 3) What will be returned when RBTreeGetSibling is called on a node with a null parent?

- A pointer to the node
- null
- A pointer to the tree's root
- Undefined/unknown



- 4) RBTreesNullOrBlack requires the node to be a leaf.

- True
- False



- 5) Which function(s) have a precondition that the node parameter must be non-null?

- All 4 functions have a precondition that the node parameter must be non-null
- RBTreeGetSibling only

©zyBooks 06/15/23 12:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



- RBTreeIsNotNullAndRed and RBTreeIsNullOrBlack
 - RBTreeGetSibling and RBTreeAreBothChildrenBlack
- 6) If RBTreeGetSibling returns a non-null, red node, then the node's parent must be non-null and black.
- True
 - False

©zyBooks 06/15/23 12:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



Prepare-for-removal algorithm overview

Preparation for removing a black node requires altering the number of black nodes along paths to preserve the black-path-length property. The `RBTreePrepareForRemoval` algorithm uses 6 utility functions that analyze the tree and make appropriate alterations when each of the 6 cases is encountered. The utility functions return true if the case is encountered, and false otherwise. If case 1, 3, or 4 is encountered, `RBTreePrepareForRemoval` will return after calling the utility function. If case 2, 5, or 6 is encountered, additional cases must be checked.

Figure 8.4.8: `RBTreePrepareForRemoval` pseudocode.

©zyBooks 06/15/23 12:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

```
RBTreePrepareForRemoval(tree, node) {
    if (RBTreeTryCase1(tree, node))
        return

    sibling = RBTreeGetSibling(node)
    if (RBTreeTryCase2(tree, node,
    sibling))
        sibling = RBTreeGetSibling(node)
    if (RBTreeTryCase3(tree, node,
    sibling))
        sibling = RBTreeGetSibling(node)
    if (RBTreeTryCase4(tree, node,
    sibling))
        return
    if (RBTreeTryCase5(tree, node,
    sibling))
        sibling = RBTreeGetSibling(node)
    if (RBTreeTryCase6(tree, node,
    sibling))
        sibling = RBTreeGetSibling(node)

    sibling->color = node->parent->color
    node->parent->color = black
    if (node == node->parent->left) {
        sibling->right->color = black
        RBTreeRotateLeft(tree,
    node->parent)
    }
    else {
        sibling->left->color = black
        RBTreeRotateRight(tree,
    node->parent)
    }
}
```

©zyBooks 06/15/23 12:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY**8.4.3: Prepare-for-removal algorithm.**

- 1) If the condition for any of the first 6 cases is met, then an adjustment specific to the case is made and the algorithm returns without processing any additional cases.

True

False

- 2) Why is no preparation action required if the node is red?



©zyBooks 06/15/23 12:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- A red node will never have children
 - A red node will never be the root of the tree
 - A red node always has a black parent node
 - Removing a red node will not change the number of black nodes along any path
- 3) Re-computation of the sibling node after case RBTreeTryCase2, RBTreeTryCase5, or RBTreeTryCase6 implies that these functions may be doing what?
- Recoloring the node or the node's parent
 - Recoloring the node's uncle or the node's sibling
 - Rotating at one of the node's children
 - Rotating at the node's parent or the node's sibling
- 4) RBTreePrepareForRemoval performs the check `node->parent == null` on the first line. What other check is equivalent and could be used in place of the code `node->parent == null`?

- `tree->root == null`
- `tree->root == node`
- `node->color == black`

node->color == red

Prepare-for-removal algorithm cases

Preparation for removing a node first checks for each of the six cases, performing the operations below.

©zyBooks 06/15/23 12:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

1. If the node is red or the node's parent is null, then return.
2. If the node has a red sibling, then color the parent red and the sibling black. If the node is the parent's left child then rotate left at the parent, otherwise rotate right at the parent. Continue to the next step.
3. If the node's parent is black and both children of the node's sibling are black, then color the sibling red, recursively call on the node's parent, and return.
4. If the node's parent is red and both children of the node's sibling are black, then color the parent black, color the sibling red, then return.
5. If the sibling's left child is red, the sibling's right child is black, and the node is the left child of the parent, then color the sibling red and the left child of the sibling black. Then rotate right at the sibling and continue to the next step.
6. If the sibling's left child is black, the sibling's right child is red, and the node is the right child of the parent, then color the sibling red and the right child of the sibling black. Then rotate left at the sibling and continue to the next step.
7. Color the sibling the same color as the parent and color the parent black.
8. If the node is the parent's left child, then color the sibling's right child black and rotate left at the parent. Otherwise color the sibling's left child black and rotate right at the parent.

Table 8.4.1: Prepare-for-removal algorithm case descriptions.

Case #	Condition	Action if condition is true	Process additional cases after action?
1	Node is red or node's parent is null.	None.	No ©zyBooks 06/15/23 12:54 1692462 Taylor Larrechea
2	Sibling node is red.	Color parent red and sibling black. If node is left child of parent, rotate left at parent node, otherwise rotate right at parent node.	Yes COLORADOCSPB2270Summer2023
3	Parent is black and both of sibling's children are	Color sibling red and call removal preparation function on parent.	No

	black.		
4	Parent is red and both of sibling's children are black.	Color parent black and sibling red.	No
5	Sibling's left child is red, sibling's right child is black, and node is left child of parent.	Color sibling red and sibling's left child black. Rotate right at sibling.	Yes
6	Sibling's left child is black, sibling's right child is red, and node is right child of parent.	Color sibling red and sibling's right child black. Rotate left at sibling.	Yes

Table 8.4.2: Prepare-for-removal algorithm case code.

Case #	Code
1	<pre>RBTtreeTryCase1(tree, node) { if (node->color == red node->parent == null) return true else return false // not case 1 }</pre>
2	<pre>RBTtreeTryCase2(tree, node, sibling) { if (sibling->color == red) { node->parent->color = red sibling->color = black if (node == node->parent->left) RBTtreeRotateLeft(tree, node->parent) else RBTtreeRotateRight(tree, node->parent) return true } return false // not case 2 }</pre>

```
3     RBTreeTryCase3(tree, node, sibling) {
        if (node->parent->color == black &&
            RBTreeAreBothChildrenBlack(sibling)) {
            sibling->color = red
            RBTreePrepareForRemoval(tree,
node->parent)
            return true
        }
        return false // not case 3
    }
```

©zyBooks 06/15/23 12:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

```
4     RBTreeTryCase4(tree, node, sibling) {
        if (node->parent->color == red &&
            RBTreeAreBothChildrenBlack(sibling)) {
            node->parent->color = black
            sibling->color = red
            return true
        }
        return false // not case 4
    }
```

```
5     RBTreeTryCase5(tree, node, sibling) {
        if (RBTreeIsNonNullAndRed(sibling->left) &&
            RBTreeIsNullOrBlack(sibling->right) &&
            node == nodeparent->left) {
            sibling->color = red
            sibling->left->color = black
            RBTreeRotateRight(tree, sibling)
            return true
        }
        return false // not case 5
    }
```

```
6     RBTreeTryCase6(tree, node, sibling) {
        if (RBTreeIsNullOrBlack(sibling->left) &&
            RBTreeIsNonNullAndRed(sibling->right) &&
            node == node->parent->right) {
            sibling->color = red
            sibling->right->color = black
            RBTreeRotateLeft(tree, sibling)
            return true
        }
        return false // not case 6
    }
```

©zyBooks 06/15/23 12:54 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



Animation content:

undefined

Animation captions:

1. In the above tree, all paths from root to null leaves have 3 black nodes.
2. Preparation for removal of node 62 encounters case 4, since the node's parent is red and both children of the sibling are black (null).
3. The parent is colored black and the sibling is colored red.
4. The preparation leaves the tree in a state where node 62 can be removed and all red-black tree requirements would be met.

PARTICIPATION ACTIVITY

8.4.5: Removal preparation for a node can encounter more than 1 case.



Animation content:

undefined

Animation captions:

1. Preparation for removal of node 75 first encounters case 2 in RBTreePrepareForRemoval.
2. After making alterations for case 2, the code proceeds to additional case checks, ending after case 4 alterations.
3. In the resulting tree, node 75 can be removed via BSTRemove and all red-black tree requirements will hold.

PARTICIPATION ACTIVITY

8.4.6: Prepare-for-removal algorithm cases.



If unable to drag and drop, refresh the page.

RBTreeTryCase2

RBTreeTryCase4

RBTreeTryCase6

RBTreeTryCase5

RBTreeTryCase1

RBTreeTryCase3

©zyBooks 06/15/23 12:54 1692462

Taylor Larrechea

COLORADO CSPB2270Summer2023

This case function always returns true if passed a node with a red sibling.

This case function finishes preparation exclusively by recoloring nodes.

This case function never returns true if the node is the right child of the node's parent.

This case function never alters the tree.

©zyBooks 06/15/23 12:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

When this case function returns true, a left rotation at the node's sibling will have just taken place.

This case function recursively calls RBTreePrepareForRemoval if the node's parent and both children of the node's sibling are black.

Reset

©zyBooks 06/15/23 12:54 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

9.1 Constant time operations

Constant time operations

In practice, designing an efficient algorithm aims to lower the amount of time that an algorithm runs. However, a single algorithm can always execute more quickly on a faster processor. Therefore, the theoretical analysis of an algorithm describes runtime in terms of number of constant time operations, not nanoseconds. A **constant time operation** is an operation that, for a given processor, always operates in the same amount of time, regardless of input values.

PARTICIPATION ACTIVITY

9.1.1: Constant time vs. non-constant time operations.



Animation content:

undefined

Animation captions:

- Statements $x = 10$, $y = 20$, $a = 1000$, and $b = 2000$ assign values to fixed-size integer variables. Each assignment is a constant time operation.
- A CPU multiplies values 10 and 20 at the same speed as 1000 and 2000. Multiplication of fixed-size integers is a constant time operation.
- A loop that iterates x times, adding y to a sum each iteration, will take longer if x is larger. The loop is not constant time.
- String concatenation is another common operation that is not constant time, because more characters must be copied for larger strings.

PARTICIPATION ACTIVITY

9.1.2: Constant time operations.



- The statement below that assigns x with y is a constant time operation.

```
y = 10  
x = y
```

- True
- False

- A loop is never a constant time operation.

©zyBooks 06/19/23 21:17 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



- True
- False

3) The 3 constant time operations in the code below can collectively be considered 1 constant time operation.

```
x = 26.5
y = 15.5
z = x + y
```

- True
- False

©zyBooks 06/19/23 21:17 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Identifying constant time operations

The programming language being used, as well as the hardware running the code, both affect what is and what is not a constant time operation. Ex: Most modern processors perform arithmetic operations on integers and floating point values at a fixed rate that is unaffected by operand values. Part of the reason for this is that the floating point and integer values have a fixed size. The table below summarizes operations that are generally considered constant time operations.

Table 9.1.1: Common constant time operations.

Operation	Example
Addition, subtraction, multiplication, and division of fixed size integer or floating point values.	<pre>w = 10.4 x = 3.4 y = 2.0 z = (w - x) / y</pre>
Assignment of a reference, pointer, or other fixed size data value.	<pre>x = 1000 y = x a = true b = a</pre> <p>©zyBooks 06/19/23 21:17 1692462 Taylor Larrechea COLORADOCSPB2270Summer2023</p>
Comparison of two fixed size data values.	<pre>a = 100 b = 200 if (b > a) { ... }</pre>

Read or write an array element at a particular index.

```
x = arr[index]
arr[index + 1] =
x + 1
```

PARTICIPATION ACTIVITY

9.1.3: Identifying constant time operations.

©zyBooks 06/19/23 21:17 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

- 1) In the code below, suppose str1 is a pointer or reference to a string. The code only executes in constant time if the assignment copies the pointer/reference, and not all the characters in the string.

```
str2 = str1
```

- True
 False

- 2) Certain hardware may execute division more slowly than multiplication, but both may still be constant time operations.

- True
 False

- 3) The hardware running the code is the only thing that affects what is and what is not a constant time operation.

- True
 False

9.2 Growth of functions and complexity

©zyBooks 06/19/23 21:17 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Upper and lower bounds

An algorithm with runtime complexity $T(N)$ has a lower bound and an upper bound.

- **Lower bound:** A function $f(N)$ that is \leq the best case $T(N)$, for all values of $N \geq 1$.

- **Upper bound:** A function $f(N)$ that is \geq the worst case $T(N)$, for all values of $N \geq 1$.

Given a function $T(N)$, an infinite number of lower bounds and upper bounds exist. Ex: If an algorithm's best case runtime is $T(N) = 5N + 4$, then subtracting any nonnegative integer yields a lower bound: $5N + 3$, $5N + 2$, and so on. So two additional criteria are commonly used to choose a preferred upper or lower bound. The preferred bound:

1. is a single-term polynomial and
2. bounds $T(N)$ as tightly as possible.

©zyBooks 06/19/23 21:17 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Upper and lower bounds in the context of runtime complexity

This section presents upper and lower bounds specifically in the context of algorithm complexity analysis. The constraint $N \geq 1$ is included because of the assumption that every algorithm presented in this book operates on a dataset with at least 1 item.

PARTICIPATION ACTIVITY

9.2.1: Upper and lower bounds.



Animation content:

undefined

Animation captions:

1. An algorithm's worst and best case runtimes are represented by the blue and purple curves, respectively.
2. The best case expression itself is a lower bound, but is a polynomial with three terms: N^2 , N , and 1 . A single-term polynomial would provide a simpler picture.
3. N^2 , shown in yellow, is a lower bound. The lower bound is less than or equal to the best case $T(N)$ for all $N \geq 1$.
4. The worst case's highest power of N is N^2 . So the upper bound must be some constant times N^2 such that $N^2 \leq C N^2$ for all $N \geq 1$.
5. N^2 does not work. Ex: When $N = 1$, $N^2 = 1$ and $C N^2 = C$.
6. The lowest C that satisfies requirements is 30. $C N^2 \geq T(N)$ is greater than or equal to the worst case $T(N)$ for all $N \geq 1$. So $C N^2$, shown in orange, is an upper bound.
7. Together, the upper and lower bounds enclose all possible runtimes for this algorithm.

PARTICIPATION ACTIVITY

9.2.2: Upper and lower bounds.

Suppose an algorithm's best case runtime complexity is _____, and the algorithm's worst case runtime is _____.

1) The algorithm has _____.

- only one possible lower bound
- multiple, but finite, lower bounds
- an infinite number of lower bounds

2) Which is the preferred lower bound?

-
-
-

3) _____ is _____ for the algorithm.

- an upper bound
- a lower bound
- neither a lower bound nor an upper bound

4) Which function is an upper bound for the algorithm?

-
-
-

©zyBooks 06/19/23 21:17 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Growth rates and asymptotic notations

An additional simplification can factor out the constant from a bounding function, leaving a function that categorizes the algorithm's growth rate. Ex: Instead of saying that an algorithm's runtime function has an upper bound of _____, the algorithm could be described as having a worst case growth rate of _____.

. **Asymptotic notation** is the classification of runtime complexity that uses functions that indicate only the growth rate of a bounding function. Three asymptotic notations are commonly used in complexity analysis:

- **O notation** provides a growth rate for an algorithm's upper bound.
- **Ω notation** provides a growth rate for an algorithm's lower bound.
- **Θ notation** provides a growth rate that is both an upper and lower bound.

Table 9.2.1: Notations for algorithm complexity analysis.

Notation	General form	Meaning
		A positive constant exists such that, for all $N \geq 1$,
		A positive constant exists such that, for all $N \geq 1$,
		and

PARTICIPATION ACTIVITY

9.2.3: Asymptotic notations.



Suppose

1)

- True
 False



2)

- True
 False



3)

- True
 False



4)

- True
 False

©zyBooks 06/19/23 21:17 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



5)

- True
 False



9.3 O notation

Big O notation

Big O notation is a mathematical way of describing how a function (running time of an algorithm) generally behaves in relation to the input size. In Big O notation, all functions that have the same growth rate (as determined by the highest order term of the function) are characterized using the same Big O notation. In essence, all functions that have the same growth rate are considered equivalent in Big O notation.

Given a function that describes the running time of an algorithm, the Big O notation for that function can be determined using the following rules:

1. If $f(N)$ is a sum of several terms, the highest order term (the one with the fastest growth rate) is kept and others are discarded.
2. If $f(N)$ has a term that is a product of several factors, all constants (those that are not in terms of N) are omitted.

PARTICIPATION ACTIVITY

9.3.1: Determining Big O notation of a function.



Animation captions:

1. Determine a function that describes the running time of the algorithm, and then compute the Big O notation of that function.
2. Apply rules to obtain the Big O notation of the function.
3. All functions with the same growth rate are considered equivalent in Big O notation.

PARTICIPATION ACTIVITY

9.3.2: Big O notation.



- 1) Which of the following Big O notations is equivalent to $O(N+9999)$?

- $O(1)$
- $O(N)$
- $O(9999)$

©zyBooks 06/19/23 21:17 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 2) Which of the following Big O notations is equivalent to $O(734 \cdot N)$?

- $O(N)$



- O(734)
- O(734·N)

3) Which of the following Big O notations is equivalent to $O(12 \cdot N + 6 \cdot N + 1000)$?

- O(1000)
- O(N)
- O(N)

©zyBooks 06/19/23 21:17 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Big O notation of composite functions

The following rules are used to determine the Big O notation of composite functions: c denotes a constant

Figure 9.3.1: Rules for determining Big O notation of composite functions.

Composite function	Big O notation
$c \cdot O(f(N))$	$O(f(N))$
$c + O(f(N))$	$O(f(N))$
$g(N) \cdot O(f(N))$	$O(g(N) \cdot f(N))$
$g(N) + O(f(N))$	$O(g(N) + f(N))$

PARTICIPATION ACTIVITY

9.3.3: Big O notation for composite functions.

©zyBooks 06/19/23 21:17 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Determine the simplified Big O notation.

1) $10 \cdot O(N)$

- O(10)
- O(N)

O($10 \cdot N$)

2) $10 + O(N)$



O(10)

O(N)

O($10 + N$)

©zyBooks 06/19/23 21:17 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

3) $3 \cdot N \cdot O(N)$

O(N)

O($3 \cdot N$)

O(N)

4) $2 \cdot N + O(N)$



O(N)

O(N)

O($N + N$)

5)



O(\quad)

O(\quad)

O(\quad)

Runtime growth rate

One consideration in evaluating algorithms is that the efficiency of the algorithm is most critical for large input sizes. Small inputs are likely to result in fast running times because N is small, so efficiency is less of a concern. The table below shows the runtime to perform $f(N)$ instructions for different functions f and different values of N . For large N , the difference in computation time varies greatly with the rate of growth of the function f . The data assumes that a single instruction takes 1 μs to execute.

Table 9.3.1: Growth rates for different input sizes.

©zyBooks 06/19/23 21:17 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Function	$N = 10$	$N = 50$	$N = 100$	$N = 1000$	$N = 10000$	$N = 100000$
	3.3 μs	5.65 μs	6.6 μs	9.9 μs	13.3 μs	16.6 μs

	10 μ s	50 μ s	100 μ s	1000 μ s	10 ms	100 ms
	.03 ms	.28 ms	.66 ms	.0099 s	.132 s	1.66 s
	.1 ms	2.5 ms	10 ms	1 s	100 s	2.7 hours
	1 ms	.125 s	1 s	16.7 min	11.57 days	31.7 years
	.001 s	35.7 years				> 1000 years

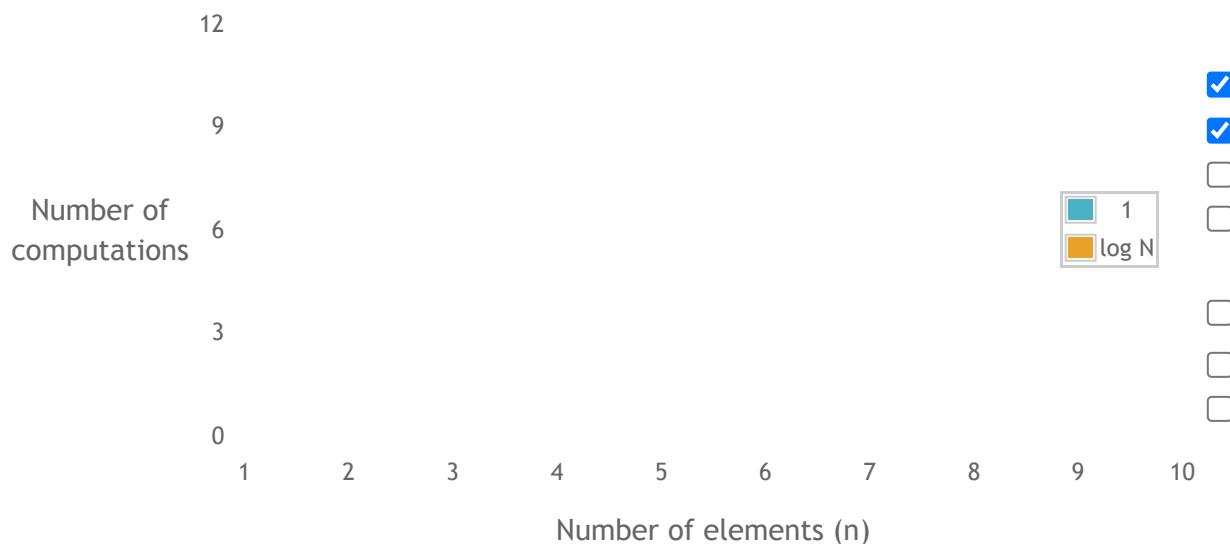
The interactive tool below illustrates graphically the growth rate of commonly encountered functions.

PARTICIPATION ACTIVITY

9.3.4: Computational complexity graphing tool.



Number of computations vs number of elements



Common Big O complexities

©zyBooks 06/19/23 21:17 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Many commonly used algorithms have running time functions that belong to one of a handful of growth functions. These common Big O notations are summarized in the following table. The table shows the Big O notation, the common word used to describe algorithms that belong to that notation, and an example with source code. Clearly, the best algorithm is one that has constant time complexity. Unfortunately, not all problems can be solved using constant complexity algorithms. In

fact, in many cases, computer scientists have proven that certain types of problems can only be solved using quadratic or exponential algorithms.

Figure 9.3.2: Runtime complexities for various code examples.

Notation	Name	Example pseudocode
O(1)	Constant	<pre>FindMin(x, y) { if (x < y) { return x } else { return y } }</pre>
O(log N)	Logarithmic	<pre>BinarySearch(numbers, N, key) { mid = 0 low = 0 high = N - 1 while (high >= low) { mid = (high + low) / 2 if (numbers[mid] < key) { low = mid + 1 } else if (numbers[mid] > key) { high = mid - 1 } else { return mid } } return -1 // not found }</pre>
O(N)	Linear	<pre>LinearSearch(numbers, numbersSize, key) { for (i = 0; i < numbersSize; ++i) { if (numbers[i] == key) { return i } } return -1 // not found }</pre>

O(N log N)	Linearithmic	<pre>MergeSort(numbers, i, k) { j = 0 if (i < k) { j = (i + k) / 2 // Find midpoint MergeSort(numbers, i, j) // Sort left part MergeSort(numbers, j + 1, k) // Sort right part Merge(numbers, i, j, k) } }</pre>
O(N)	Quadratic	<pre>SelectionSort(numbers, numbersSize) { for (i = 0; i < numbersSize; ++i) { indexSmallest = i for (j = i + 1; j < numbersSize; ++j) { if (numbers[j] < numbers[indexSmallest]) indexSmallest = j } temp = numbers[i] numbers[i] = numbers[indexSmallest] numbers[indexSmallest] = temp } }</pre>
O(c)	Exponential	<pre>Fibonacci(N) { if ((1 == N) (2 == N)) { return 1 } return Fibonacci(N-1) + Fibonacci(N-2) }</pre>

PARTICIPATION ACTIVITY

9.3.5: Big O notation and growth rates.



1) O(5) has a ____ runtime complexity.

©zyBooks 06/19/23 21:17 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- constant
- linear
- exponential

2) O(N log N) has a ____ runtime complexity.



- constant
- linearithmic
- logarithmic

3) $O(N + N)$ has a ____ runtime complexity.

©zyBooks 06/19/23 21:17 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- linear-quadratic
- exponential
- quadratic

4) A linear search has a ____ runtime complexity.

- $O(\log N)$
- $O(N)$
- $O(N)$

5) A selection sort has a ____ runtime complexity.

- $O(N)$
- $O(N \log N)$
- $O(N)$

9.4 Algorithm analysis

Worst-case algorithm analysis

To analyze how runtime of an algorithm scales as the input size increases, we first determine how many operations the algorithm executes for a specific input size, N . Then, the big-O notation for that function is determined. Algorithm runtime analysis often focuses on the worst-case runtime complexity. The **worst-case runtime** of an algorithm is the runtime complexity for an input that results in the longest execution. Other runtime analyses include best-case runtime and average-case runtime. Determining the average-case runtime requires knowledge of the statistical properties of the expected data inputs.

PARTICIPATION ACTIVITY

9.4.1: Runtime analysis: Finding the max value.

**Animation captions:**

1. Runtime analysis determines the total number of operations. Operations include assignment, addition, comparison, etc.
2. The for loop iterates N times, but the for loop's initial expression $i = 0$ is executed once.
3. For each loop iteration, the increment and comparison expressions are each executed once. In the worst-case, the if's expression is true, resulting in 2 operations.
4. One additional comparison is made before the loop ends.
5. The function $f(N)$ specifies the number of operations executed for input size N. The big-O notation for the function is the algorithm's worst-case runtime complexity.

PARTICIPATION ACTIVITY

9.4.2: Worst-case runtime analysis.



- 1) Which function best represents the number of operations in the worst-case?



```
i = 0
sum = 0
while (i < N) {
    sum = sum + numbers[i]
    ++i
}
```

- $f(N) = 3N + 2$
- $f(N) = 3N + 3$
- $f(N) = 2 + N(N + 1)$

- 2) What is the big-O notation for the worst-case runtime?



```
negCount = 0
for(i = 0; i < N; ++i) {
    if (numbers[i] < 0) {
        ++negCount
    }
}
```

©zyBooks 06/19/23 21:17 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- $f(N) = 2 + 4N + 1$
- $O(4N + 3)$
- $O(N)$



- 3) What is the big-O notation for the worst-case runtime?

```
for (i = 0; i < N; ++i) {
    if ((i % 2) == 0) {
        outVal[i] = inVals[i] * i
    }
}
```

- O(1)
- O(∞)
- O(N)

©zyBooks 06/19/23 21:17 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

- 4) Assuming nVal is an integer, what is the big-O notation for the worst-case runtime?

```
nVal = N
steps = 0
while (nVal > 0) {
    nVal = nVal / 2
    steps = steps + 1
}
```

- O(log N)
- O(∞)
- O(N)

- 5) What is the big-O notation for the *best*-case runtime?

```
i = 0
belowThresholdSum = 0.0
belowThresholdCount = 0
while (i < N && numbers[i] <=
threshold) {
    belowThresholdCount += 1
    belowThresholdSum +=
numbers[i]
    i += 1
}
avgBelow = belowThresholdSum /
belowThresholdCount
```

- O(1)
- O(N)

©zyBooks 06/19/23 21:17 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Counting constant time operations

For algorithm analysis, the definition of a single operation does not need to be precise. An operation can be any statement (or constant number of statements) that has a constant runtime complexity, $O(1)$. Since constants are omitted in big-O notation, any constant number of constant time operations is $O(1)$. So, precisely counting the number of constant time operations in a finite sequence is not needed. Ex: An algorithm with a single loop that execute 5 operations before the loop, 3 operations each loop iteration, and 6 operations after the loop would have a runtime of $f(N) = 5 + 3N + 6$, which can be written as $O(1) + O(N) + O(1) = O(N)$. If the number of operations before the loop was 100, the big-O notation for those operations is still $O(1)$.

©zyBooks 06/19/23 21:17 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

9.4.3: Simplified runtime analysis: A constant number of constant time operations is $O(1)$.



Animation captions:

1. Constants are omitted in big-O notation, so any constant number of constant time operations is $O(1)$.
2. The for loop iterates N times. Big-O complexity can be written as a composite function and simplified.

PARTICIPATION ACTIVITY

9.4.4: Constant time operations.



- 1) A for loop of the form `for (i = 0; i < N; ++i) {}` that does not have nested loops or function calls, and does not modify `i` in the loop will always have a complexity of $O(N)$.

- True
- False

- 2) The complexity of the algorithm below is $O(1)$.



```
if (timeHour < 6) {  
    tollAmount = 1.55  
}  
else if (timeHour < 10) {  
    tollAmount = 4.65  
}  
else if (timeHour < 18) {  
    tollAmount = 2.35  
}  
else {  
    tollAmount = 1.55  
}
```

©zyBooks 06/19/23 21:17 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

True False

- 3) The complexity of the algorithm below is O(1). 

©zyBooks 06/19/23 21:17 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

```
for (i = 0; i < 24; ++i) {
    if (timeHour < 6) {
        tollSchedule[i] = 1.55
    }
    else if (timeHour < 10) {
        tollSchedule[i] = 4.65
    }
    else if (timeHour < 18) {
        tollSchedule[i] = 2.35
    }
    else {
        tollSchedule[i] = 1.55
    }
}
```

 True False

Runtime analysis of nested loops

Runtime analysis for nested loops requires summing the runtime of the inner loop over each outer loop iteration. The resulting summation can be simplified to determine the big-O notation.

PARTICIPATION ACTIVITY

 9.4.5: Runtime analysis of nested loop: Selection sort algorithm. 

Animation content:

undefined

Animation captions:

©zyBooks 06/19/23 21:17 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

1. For each iteration of the outer loop, the runtime of the inner loop is determined and added together to form a summation. For iteration $i = 0$, the inner loop executes $N - 1$ iterations.
2. For $i = 1$, the inner loop iterates $N - 2$ times: iterating from $j = 2$ to $N - 1$.
3. For $i = N - 3$, the inner loop iterates twice: iterating from $j = N - 2$ to $N - 1$. For $i = N - 2$, the inner loop iterates once: iterating from $j = N - 1$ to $N - 1$.
4. For $i = N - 1$, the inner loop iterates 0 times. The summation is the sum of a consecutive sequence of numbers from $N - 1$ to 0.

5. The sequence contains $N / 2$ pairs, each summing to $N - 1$, and can be simplified.
6. Each iteration of the loops requires a constant number of operations, which is defined as the constant c .
7. Additionally, each iteration of the outer loop requires a constant number of operations, which is defined as the constant d .
8. Big-O notation omits the constant values, and the runtime is equal to the summation of the total inner loop iterations.

©zyBooks 06/19/23 21:17 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Figure 9.4.1: Common summation: Summation of consecutive numbers.

PARTICIPATION ACTIVITY

9.4.6: Nested loops.



Determine the big-O worst-case runtime for each algorithm.

1) `for (i = 0; i < N; i++) {
 for (j = 0; j < N; j++) {
 if (numbers[i] <
 numbers[j]) {
 ++eqPerms
 }
 else {
 ++neqPerms
 }
 }
}`

 $O(N)$ $O(N^2)$ 

2) `for (i = 0; i < N; i++) {
 for (j = 0; j < (N - 1);
 j++) {
 if (numbers[j + 1] <
 numbers[j]) {
 temp = numbers[j]
 numbers[j] = numbers[j
+ 1]
 numbers[j + 1] = temp
 }
 }
}`



©zyBooks 06/19/23 21:17 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

- O(N)
- O(N)

3) `for (i = 0; i < N; i = i + 2)
{
 for (j = 0; j < N; j = j +
2) {
 cVals[i][j] = inVals[i] *
j
 }
}`



©zyBooks 06/19/23 21:17 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- O(N)
- O(N)

4) `for (i = 0; i < N; ++i) {
 for (j = i; j < N - 1; ++j)
 {
 cVals[i][j] = inVals[i] *
j
 }
}`



- O(N)
- O(N)

5) `for (i = 0; i < N; ++i) {
 sum = 0
 for (j = 0; j < N; ++j) {
 for (k = 0; k < N; ++k) {
 sum = sum + aVals[i]
[k] * bVals[k][j]
 }

 cVals[i][j] = sum
 }
}`



- O(N)
- O(N)
- O(N)

©zyBooks 06/19/23 21:17 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

9.5 Searching and algorithms



This section has been set as optional by your instructor.

Algorithms

An **algorithm** is a sequence of steps for accomplishing a task. **Linear search** is a search algorithm that starts from the beginning of a list, and checks each element until the search key is found or the end of the list is reached.

PARTICIPATION
ACTIVITY

9.5.1: Linear search algorithm checks each element until key is found.

©zyBooks 06/19/23 21:17 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Animation content:

undefined

Animation captions:

1. Linear search starts at first element and searches elements one-by-one.
2. Linear search will compare all elements if the search key is not present.

Figure 9.5.1: Linear search algorithm.

©zyBooks 06/19/23 21:17 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

```
LinearSearch(numbers, numbersSize, key) {  
    i = 0  
  
    for (i = 0; i < numbersSize; ++i) {  
        if (numbers[i] == key) {  
            return i  
        }  
    }  
  
    return -1 // not found  
}  
  
main() {  
    numbers = {2, 4, 7, 10, 11, 32, 45, 87}  
    NUMBERS_SIZE = 8  
    i = 0  
    key = 0  
    keyIndex = 0  
  
    print("NUMBERS: ")  
    for (i = 0; i < NUMBERS_SIZE; ++i) {  
        print(numbers[i] + " ")  
    }  
    printLine()  
  
    print("Enter a value: ")  
    key = getIntFromUser()  
  
    keyIndex = LinearSearch(numbers, NUMBERS_SIZE, key)  
  
    if (keyIndex == -1) {  
        printLine(key + " was not found.")  
    }  
    else {  
        printLine("Found " + key + " at index " + keyIndex + ".")  
    }  
}
```

©zyBooks 06/19/23 21:17 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

```
NUMBERS: 2 4 7 10 11 32 45 87  
Enter a value: 10  
Found 10 at index 3.  
...  
NUMBERS: 2 4 7 10 11 32 45 87  
Enter a value: 17  
17 was not found.
```

©zyBooks 06/19/23 21:17 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

9.5.2: Linear search algorithm execution.



Given list: (20, 4, 114, 23, 34, 25, 45, 66, 77, 89, 11).

- 1) How many list elements will be compared to find 77 using linear search?



//**Show answer**

- 2) How many list elements will be checked to find the value 114 using linear search?

 //**Check****Show answer**

©zyBooks 06/19/23 21:17 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



- 3) How many list elements will be checked if the search key is not found using linear search?

 //**Check****Show answer**

Algorithm runtime

An algorithm's **runtime** is the time the algorithm takes to execute. If each comparison takes 1 μ s (1 microsecond), a linear search algorithm's runtime is up to 1 s to search a list with 1,000,000 elements, 10 s for 10,000,000 elements, and so on. Ex: Searching Amazon's online store, which has more than 200 million items, could require more than 3 minutes.

An algorithm typically uses a number of steps proportional to the size of the input. For a list with 32 elements, linear search requires at most 32 comparisons: 1 comparison if the search key is found at index 0, 2 if found at index 1, and so on, up to 32 comparisons if the search key is not found. For a list with N elements, linear search thus requires at most N comparisons. The algorithm is said to require "on the order" of N comparisons.

©zyBooks 06/19/23 21:17 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

**PARTICIPATION ACTIVITY**

9.5.3: Linear search runtime.

- 1) Given a list of 10,000 elements, and if each comparison takes 2 μ s, what is the fastest possible runtime for linear search?



/ / μs

Check**Show answer**

- 2) Given a list of 10,000 elements, and if each comparison takes 2 μs , what is the longest possible runtime for linear search?

/ / μs

Check**Show answer**

©zyBooks 06/19/23 21:17 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



9.6 Analyzing the time complexity of recursive algorithms



This section has been set as optional by your instructor.

Recurrence relations

The runtime complexity $T(N)$ of a recursive function will have function T on both sides of the equation. Ex: Binary search performs constant time operations, then a recursive call that operates on half of the input, making the runtime complexity $T(N) = O(1) + T(N / 2)$. Such a function is known as a **recurrence relation**: A function $f(N)$ that is defined in terms of the same function operating on a value $< N$.

Using O -notation to express runtime complexity of a recursive function requires solving the recurrence relation. For simpler recursive functions such as binary search, runtime complexity can be determined by expressing the number of function calls as a function of N .

Taylor Larrechea
COLORADOCSPB2270Summer2023**PARTICIPATION ACTIVITY**

9.6.1: Worst case binary search runtime complexity.



Animation content:

undefined

Animation captions:

1. In the non-base case, BinarySearch does some $O(1)$ operations plus a recursive call on half the input list.
2. The maximum number of recursive calls can be computed for any known input size. For size 1, 1 recursive call is made.
3. Additional entries in the table can be filled. A list of size 32 is split in half 6 times before encountering the base case.
4. By analyzing the pattern, the total number of function calls can be expressed as a function of N .
5. The number of function calls corresponds to the runtime complexity.

©zyBooks 06/19/23 21:17 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

9.6.2: Binary search and recurrence relations.



- 1) When the low and high arguments are equal, BinarySearch() has 0 items to search and so immediately returns -1.

- True
- False



- 2) Suppose BinarySearch() is used to search for a key within an array with 64 numbers. If the key is not found, how many recursive calls to BinarySearch() are made?

- 1
- 7
- 64



- 3) Which function is a recurrence relation?

-
-
-



©zyBooks 06/19/23 21:17 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Recursion trees

The runtime complexity of any recursive function can be split into 2 parts: operations done directly by the function and operations done by recursive calls made by the function. Ex: For binary search's $T(N) = O(1) + T(N / 2)$, $O(1)$ represents operations directly done by the function and $T(N / 2)$ represents operation done by a recursive call. A useful tool for solving recurrences is a **recursion tree**: A visual

diagram of an operation done by a recursive function, that separates operations done directly by the function and operations done by recursive calls.

PARTICIPATION ACTIVITY

9.6.3: Recursion trees.


Animation captions:

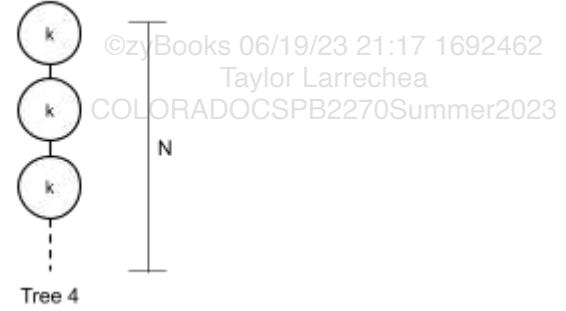
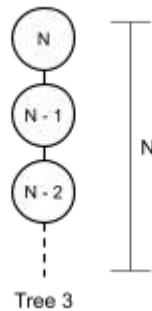
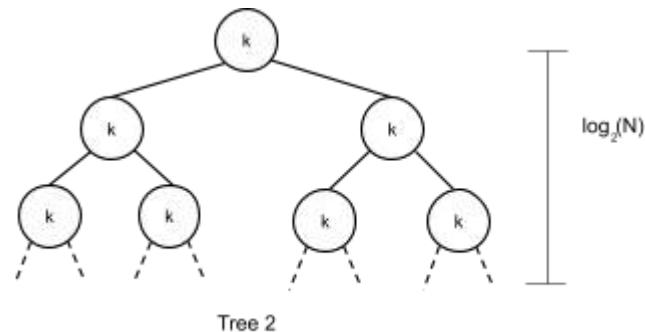
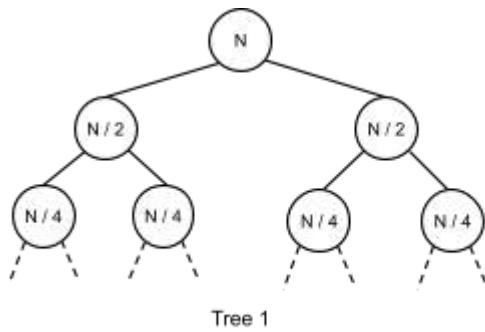
©zyBooks 06/19/23 21:17 1692462

Taylor Larrechea

1. An algorithm like binary search does a constant number of operations, k , followed by a recursive call on half the list.
2. The root node in the recursion tree represents k operations inside the first function call.
3. Recursive operations are represented below the node. The first recursive call also does k operations.
4. The tree's height corresponds to the number of recursive calls. Splitting the input in half each time results in recursive calls. $O(\quad) = O(\quad)$.
5. Another algorithm may perform N operations then 2 recursive calls, each on $N / 2$ items. The root node represents N operations.
6. The initial call makes 2 recursive calls, each of which has a local N value of the initial N value / 2.
7. N operations are done per level.
8. The tree has $O(\quad)$ levels. $O(\quad) = O(\quad)$ operations are done in total.

PARTICIPATION ACTIVITY

9.6.4: Matching recursion trees with runtime complexities.


 ©zyBooks 06/19/23 21:17 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

If unable to drag and drop, refresh the page.

[Tree 3](#)[Tree 4](#)[Tree 2](#)[Tree 1](#)

$$T(N) = k + T(N / 2) + T(N / 2)$$

$$T(N) = k + T(N - 1)$$

©zyBooks 06/19/23 21:17 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

$$T(N) = N + T(N - 1)$$

$$T(N) = N + T(N / 2) + T(N / 2)$$

[Reset](#)**PARTICIPATION ACTIVITY**

9.6.5: Recursion trees.



Suppose a recursive function's runtime is

- 1) How many levels will the recursion tree have?

- 7
-
-

- 2) What is the runtime complexity of the function using O notation?

- $O(1)$
- $O(\quad)$
- $O(\quad)$

PARTICIPATION ACTIVITY

9.6.6: Recursion trees.



Suppose a recursive function's runtime is

- 1) How many levels will the recursion tree have?

-
-

©zyBooks 06/19/23 21:17 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



- 2) The runtime can be expressed by the series $N + (N - 1) + (N - 2) + \dots + 3 + 2 + 1$. Which expression is mathematically equivalent?



- 3) What is the runtime complexity of the function using O notation?



$O()$

$O()$

©zyBooks 06/19/23 21:17 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

©zyBooks 06/19/23 21:17 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

10.1 Sorting: Introduction

Sorting is the process of converting a list of elements into ascending (or descending) order. For example, given a list of numbers (17, 3, 44, 6, 9), the list after sorting is (3, 6, 9, 17, 44). You may have carried out sorting when arranging papers in alphabetical order, or arranging envelopes to have ascending zip codes (as required for bulk mailings).

©zyBooks 06/21/23 22:37 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

The challenge of sorting is that a program can't "see" the entire list to know where to move an element. Instead, a program is limited to simpler steps, typically observing or swapping just two elements at a time. So sorting just by swapping values is an important part of sorting algorithms.

PARTICIPATION ACTIVITY

10.1.1: Sort by swapping tool.



Sort the numbers from smallest on left to largest on right. Select two numbers then click "Swap values".

Start

--	--	--	--	--	--	--

Swap

Time - Best time -

[Clear best](#)

PARTICIPATION ACTIVITY

10.1.2: Sorted elements.



1) The list is sorted into ascending order:

(3, 9, 44, 18, 76)

True

False

©zyBooks 06/21/23 22:37 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

2) The list is sorted into descending order:

(20, 15, 10, 5, 0)

True



False

3) The list is sorted into descending order:

(99.87, 99.02, 67.93, 44.10)

 True False

4) The list is sorted into descending order:

(F, D, C, B, A)

 True False

5) The list is sorted into ascending order:

(chopsticks, forks, knives, spork)

 True False

6) The list is sorted into ascending order:

(great, greater, greatest)

 True False

10.2 Quicksort

Quicksort

Quicksort is a sorting algorithm that repeatedly partitions the input into low and high parts (each part unsorted), and then recursively sorts each of those parts. To partition the input, quicksort chooses a pivot to divide the data into low and high parts. The **pivot** can be any value within the array being sorted, commonly the value of the middle array element. Ex: For the list (4, 34, 10, 25, 1), the middle element is located at index 2 (the middle of indices [0, 4]) and has a value of 10.

Once the pivot is chosen, the quicksort algorithm divides the array into two parts, referred to as the low partition and the high partition. All values in the low partition are less than or equal to the pivot value. All values in the high partition are greater than or equal to the pivot value. The values in each partition are not necessarily sorted. Ex: Partitioning (4, 34, 10, 25, 1) with a pivot value of 10 results in a low partition of (4, 1, 10) and a high partition of (25, 34). Values equal to the pivot may appear in either or both of the partitions.

**PARTICIPATION
ACTIVITY**

10.2.1: Quicksort partitions data into a low partition with values \leq pivot and a high partition with values \geq pivot.

**Animation content:**

Step 1: Array is shown as [7, 4, 6, 18, 8]. Labels lowIndex and highIndex point to array indices 0 and 4, respectively. Code execution begins within the Partition function, and the calculation of the midpoint is shown as:

$$\begin{aligned} \text{lowIndex} + (\text{highIndex} - \text{lowIndex}) / 2 \\ = 0 + (4 - 0) / 2 \\ = 2 \end{aligned}$$

The pivot variable is then assigned with numbers[midpoint], or 6.

Step 2: The first nested while loop executes. Since the condition $7 < 6$ is false, lowIndex is not incremented and remains 0.

Step 3: The second nested while loop executes. Conditions $6 < 8$ and $6 < 18$ are true, so highIndex is decremented twice to become 2. The condition $6 < 6$ is false, so the second nested while loop then ends.

Step 4: Elements at indices 0 and 2 (lowIndex and highIndex) are swapped, yielding the array: [6, 4, 7, 18, 8].

Step 5: Execution continues, changing lowIndex and highIndex to 1. The next iteration of the outermost loop begins. lowIndex is incremented to 2, since $4 < 6$. The condition $7 < 6$ is false, so lowIndex is not incremented further. The second nested while loop does not decrement highIndex, since the condition $6 < 4$ is false. The following if statement's condition of $\text{lowIndex} \geq \text{highIndex}$ is now true, so the variable done is assigned with true.

Step 6: The Partition() function's execution ends, returning highIndex's value of 1.

Animation captions:

1. The pivot value is the value of the middle element.
2. lowIndex is incremented until a value greater than or equal to the pivot is found.
3. highIndex is decremented until a value less than or equal to the pivot is found.
4. Elements at indices lowIndex and highIndex are swapped, moving those elements to the correct partitions.
5. The partition process repeats until indices lowIndex and highIndex reach or pass each other, indicating all elements have been partitioned.
6. Once partitioned, the algorithm returns highIndex, which is the highest index of the low partition. The partitions are not yet sorted.

Partitioning algorithm

The partitioning algorithm uses two index variables `lowIndex` and `highIndex`, initialized to the left and right sides of the current elements being sorted. As long as the value at index `lowIndex` is less than the pivot value, the algorithm increments `lowIndex`, because the element should remain in the low partition. Likewise, as long as the value at index `highIndex` is greater than the pivot value, the algorithm decrements `highIndex`, because the element should remain in the high partition. Then, if `lowIndex >= highIndex`, all elements have been partitioned, and the partitioning algorithm returns `highIndex`, which is the index of the last element in the low partition. Otherwise, the elements at indices `lowIndex` and `highIndex` are swapped to move those elements to the correct partitions. The algorithm then increments `lowIndex`, decrements `highIndex`, and repeats.

PARTICIPATION ACTIVITY

10.2.2: Quicksort pivot location and value.



Determine the midpoint and pivot values.

- 1) numbers = (1, 2, 3, 4, 5), `lowIndex` = 0, `highIndex` = 4

`midpoint` = //

Check**Show answer**

- 2) numbers = (1, 2, 3, 4, 5), `lowIndex` = 0, `highIndex` = 4

`pivot` = //

Check**Show answer**

- 3) numbers = (200, 11, 38, 9),
`lowIndex` = 0, `highIndex` = 3

`midpoint` = //

Check**Show answer**

©zyBooks 06/21/23 22:37 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



- 4) numbers = (200, 11, 38, 9),
`lowIndex` = 0, `highIndex` = 3

`pivot` = //

Check**Show answer**



- 5) numbers = (55, 7, 81, 26, 0, 34, 68, 125), lowIndex = 3, highIndex = 7

midpoint = //

Check

[Show answer](#)

- 6) numbers = (55, 7, 81, 26, 0, 34, 68, 125), lowIndex = 3, highIndex = 7

pivot = //

Check

[Show answer](#)

©zyBooks 06/21/23 22:37 169246
Taylor Larrechea
COLORADOCSPB2270Summer2023

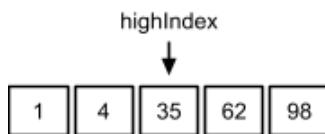
PARTICIPATION ACTIVITY

10.2.3: Low and high partitions.



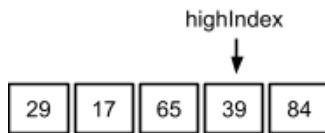
Determine if the low and high partitions are correct given highIndex and pivot.

- 1) pivot = 35



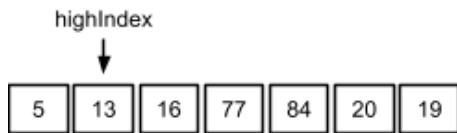
- Correct
- Incorrect

- 2) pivot = 65



- Correct
- Incorrect

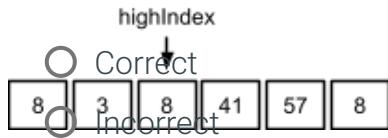
- 3) pivot = 5



- Correct
- Incorrect

- 4) pivot = 8

©zyBooks 06/21/23 22:37 169246
Taylor Larrechea
COLORADOCSPB2270Summer2023



©zyBooks 06/21/23 22:37 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Recursively sorting partitions

Once partitioned, each partition needs to be sorted. Quicksort is typically implemented as a recursive algorithm using calls to quicksort to sort the low and high partitions. This recursive sorting process continues until a partition has one or zero elements, and thus is already sorted.

PARTICIPATION
ACTIVITY

10.2.4: Quicksort.



Animation content:

undefined

Animation captions:

1. The list from low index 0 to high index 4 has more than 1 element, so Partition is called.
2. Quicksort is called recursively to sort the low and high partitions.
3. The low partition has more than one element. Partition is called for the low partition, followed by recursive calls to Quicksort.
4. Each partition that has one element is already sorted.
5. The high partition has more than one element and thus is partitioned and recursively sorted.
6. The low partition with two elements is partitioned and recursively sorted.
7. Each remaining partition with only one element is already sorted.
8. All elements are sorted.

Below is the recursive quicksort algorithm, including quicksort's key component, the partitioning function.

©zyBooks 06/21/23 22:37 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Figure 10.2.1: Quicksort algorithm.

©zyBooks 06/21/23 22:37 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

©zyBooks 06/21/23 22:37 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

```
Partition(numbers, lowIndex, highIndex) {
    // Pick middle element as pivot
    midpoint = lowIndex + (highIndex - lowIndex) / 2
    pivot = numbers[midpoint]

    done = false
    while (!done) {
        // Increment lowIndex while numbers[lowIndex] < pivot
        while (numbers[lowIndex] < pivot) {
            lowIndex += 1
        }

        // Decrement highIndex while pivot < numbers[highIndex]
        while (pivot < numbers[highIndex]) {
            highIndex -= 1
        }

        // If zero or one elements remain, then all numbers are
        // partitioned. Return highIndex.
        if (lowIndex >= highIndex) {
            done = true
        }
        else {
            // Swap numbers[lowIndex] and numbers[highIndex]
            temp = numbers[lowIndex]
            numbers[lowIndex] = numbers[highIndex]
            numbers[highIndex] = temp

            // Update lowIndex and highIndex
            lowIndex += 1
            highIndex -= 1
        }
    }

    return highIndex
}

Quicksort(numbers, lowIndex, highIndex) {
    // Base case: If the partition size is 1 or zero
    // elements, then the partition is already sorted
    if (lowIndex >= highIndex) {
        return
    }

    // Partition the data within the array. Value lowEndIndex
    // returned from partitioning is the index of the low
    // partition's last element.
    lowEndIndex = Partition(numbers, lowIndex, highIndex)

    // Recursively sort low partition (lowIndex to lowEndIndex)
    // and high partition (lowEndIndex + 1 to highIndex)
    Quicksort(numbers, lowIndex, lowEndIndex)
    Quicksort(numbers, lowEndIndex + 1, highIndex)
}

main() {
    numbers[] = { 10, 2, 78, 4, 45, 32, 7, 11 }
    NUMBERS_SIZE = 8
    i = 0
```

©zyBooks 06/21/23 22:37 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

```

print("UNSORTED: ")
for(i = 0; i < NUMBERS_SIZE; ++i) {
    print(numbers[i] + " ")
}
printLine()

// Initial call to quicksort
Quicksort(numbers, 0, NUMBERS_SIZE - 1)

print("SORTED: ")
for(i = 0; i < NUMBERS_SIZE; ++i) {
    print(numbers[i] + " ")
}
printLine()
}

```

©zyBooks 06/21/23 22:37 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

UNSORTED: 10 2 78 4 45 32 7 11
 SORTED: 2 4 7 10 11 32 45 78

Quicksort activity

The following activity helps build intuition as to how partitioning a list into two unsorted parts, one part \leq a pivot value and the other part \geq a pivot value, and then recursively sorting each part, ultimately leads to a sorted list.

PARTICIPATION
ACTIVITY

10.2.5: Quicksort tool.



Select all values in the current window that are less than the pivot for the left part, then press "Partition". If a value equals pivot, you can choose which part, but each part must contain at least one number. Light blue means current window. Green means sorted.

Start



Partition

©zyBooks 06/21/23 22:37 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

Time - Best time -

Clear best

Quicksort runtime

The quicksort algorithm's runtime is typically $O(N \log N)$. Quicksort has several partitioning levels, the first level dividing the input into 2 parts, the second into 4 parts, the third into 8 parts, etc. At each level, the algorithm does at most N comparisons moving the lowIndex and highIndex indices. If the pivot yields two equal-sized parts, then there will be $\log N$ levels, requiring the $N * \log N$ comparisons.

PARTICIPATION
ACTIVITY

10.2.6: Quicksort runtime.



Assume quicksort always chooses a pivot that divides the elements into two equal parts.

- 1) How many partitioning levels are required for a list of 8 elements?

Check

Show answer



- 2) How many partitioning levels are required for a list of 1024 elements?

Check

Show answer



- 3) How many total comparisons are required to sort a list of 1024 elements?



Check**Show answer**

©zyBooks 06/21/23 22:37 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Worst case runtime

For typical unsorted data, such equal partitioning occurs. However, partitioning may yield unequally sized parts in some cases. If the pivot selected for partitioning is the smallest or largest element, one partition will have just 1 element, and the other partition will have all other elements. If this unequal partitioning happens at every level, there will be $N - 1$ levels, yielding $N + N-1 + N-2 + \dots + 2 + 1 =$

, which is $O(N^2)$. So the worst case runtime for the quicksort algorithm is $O(N^2)$. Fortunately, this worst case runtime rarely occurs.

PARTICIPATION ACTIVITY

10.2.7: Worst case quicksort runtime.



Assume quicksort always chooses the smallest element as the pivot.

- Given numbers = (7, 4, 2, 25, 19),
lowIndex = 0, and highIndex = 4,
what are the contents of the low
partition? Type answer as: 1, 2, 3

Check**Show answer**

- How many partitioning levels are required for a list of 5 elements?

Check**Show answer**

- How many partitioning levels are required for a list of 1024 elements?

©zyBooks 06/21/23 22:37 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



Check**Show answer**

- 4) How many total calls to the Quicksort() function are made to sort a list of 1024 elements?

**Check****Show answer**

©zyBooks 06/21/23 22:37 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

CHALLENGE ACTIVITY**10.2.1: Quicksort.**

489394.3384924.qx3zqy7

Start

Given numbers = (16, 28, 39, 57, 95, 54, 45), lowIndex = 0, highIndex = 6

What is the midpoint?

 Ex: 9

What is the pivot?

1

2

3

4

5

6

Check**Next**

©zyBooks 06/21/23 22:37 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

10.3 Merge sort

Merge sort overview

Merge sort is a sorting algorithm that divides a list into two halves, recursively sorts each half, and then merges the sorted halves to produce a sorted list. The recursive partitioning continues until a list of 1 element is reached, as a list of 1 element is already sorted.

PARTICIPATION
ACTIVITY

10.3.1: Merge sort recursively divides the input into two halves, sorts each half, and merges the lists together.

©zyBooks 06/21/23 22:37 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



Animation captions:

1. MergeSort recursively divides the list into two halves.
2. The list is divided until a list of 1 element is found.
3. A list of 1 element is already sorted.
4. At each level, the sorted lists are merged together while maintaining the sorted order.

Merge sort partitioning

The merge sort algorithm uses three index variables to keep track of the elements to sort for each recursive function call. The index variable i is the index of first element in the list, and the index variable k is the index of the last element. The index variable j is used to divide the list into two halves. Elements from i to j are in the left half, and elements from $j + 1$ to k are in the right half.

PARTICIPATION
ACTIVITY

10.3.2: Merge sort partitioning.



Determine the index j and the left and right partitions.

- 1) numbers = (1, 2, 3, 4, 5), $i = 0$, $k = 4$

$j =$ //

Check

Show answer



- 2) numbers = (1, 2, 3, 4, 5), $i = 0$, $k = 4$

Left partition = (
 //)

Check

Show answer



©zyBooks 06/21/23 22:37 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

- 3) numbers = (1, 2, 3, 4, 5), $i = 0$, $k = 4$



Right partition = (//

Check**Show answer**

- 4) numbers = (34, 78, 14, 23, 8, 35), i = 3, k = 5

j = //

Check**Show answer**

- 5) numbers = (34, 78, 14, 23, 8, 35), i = 3, k = 5

Left partition = (//

Check**Show answer**

- 6) numbers = (34, 78, 14, 23, 8, 35), i = 3, k = 5

Right partition = (//

Check**Show answer**

Merge sort algorithm

Merge sort merges the two sorted partitions into a single list by repeatedly selecting the smallest element from either the left or right partition and adding that element to a temporary merged list. Once fully merged, the elements in the temporary merged list are copied back to the original list.

PARTICIPATION ACTIVITY

10.3.3: Merging partitions: Smallest element from left or right partition is added one at a time to a temporary merged list. Once merged, temporary list is copied back to the original list.

Animation content:

undefined

Animation captions:

1. Create a temporary list for merged numbers. Initialize mergePos, leftPos, and rightPos to the first element of each of the corresponding list.
2. Compare the element in the left and right partitions. Add the smallest value to the temporary list and update the relevant indices.
3. Continue to compare the elements in the left and right partitions until one of the partitions is empty.
4. If a partition is not empty, copy the remaining elements to the temporary list. The elements are already in sorted order.
5. Lastly, the elements in the temporary list are copied back to the original list.

©zyBooks 06/21/23 22:37 1692462Taylor LarrecheaCOLORADOCSPB2270Summer2023**PARTICIPATION ACTIVITY**

10.3.4: Tracing merge operation.



Trace the merge operation by determining the next value added to mergedNumbers.

14	18	35	17	38	49
0	1	2	3	4	5

- 1) leftPos = 0, rightPos = 3

**Check****Show answer**

- 2) leftPos = 1, rightPos = 3

**Check****Show answer**

- 3) leftPos = 1, rightPos = 4

**Check****Show answer**

- 4) leftPos = 2, rightPos = 4

**Check****Show answer**

- 5) leftPos = 3, rightPos = 4

©zyBooks 06/21/23 22:37 1692462Taylor LarrecheaCOLORADOCSPB2270Summer2023

**Check****Show answer**

- 6) $\text{leftPos} = 3, \text{rightPos} = 5$

**Check****Show answer**

©zyBooks 06/21/23 22:37 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



Figure 10.3.1: Merge sort algorithm.

©zyBooks 06/21/23 22:37 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

©zyBooks 06/21/23 22:37 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

©zyBooks 06/21/23 22:37 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

```
Merge(numbers, i, j, k) {
    mergedSize = k - i + 1
    mergePos = 0
    leftPos = 0
partition
    rightPos = 0
partition
    mergedNumbers = new int[mergedSize] // Dynamically allocates temporary
array
                                                // for merged numbers
                                                @zyBooks 06/21/23 22:37 1692462
                                                Taylor Larrechea
                                                COLORADOCSPB2270Summer2023

    leftPos = i // Initialize left partition
position
    rightPos = j + 1 // Initialize right partition
position

    // Add smallest element from left or right partition to merged numbers
    while (leftPos <= j && rightPos <= k) {
        if (numbers[leftPos] <= numbers[rightPos]) {
            mergedNumbers[mergePos] = numbers[leftPos]
            ++leftPos
        }
        else {
            mergedNumbers[mergePos] = numbers[rightPos]
            ++rightPos

        }
        ++mergePos
    }

    // If left partition is not empty, add remaining elements to merged
numbers
    while (leftPos <= j) {
        mergedNumbers[mergePos] = numbers[leftPos]
        ++leftPos
        ++mergePos
    }

    // If right partition is not empty, add remaining elements to merged
numbers
    while (rightPos <= k) {
        mergedNumbers[mergePos] = numbers[rightPos]
        ++rightPos
        ++mergePos
    }

    // Copy merge number back to numbers
    for (mergePos = 0; mergePos < mergedSize; ++mergePos) {
        numbers[i + mergePos] = mergedNumbers[mergePos]
    }
}

MergeSort(numbers, i, k) {
    j = 0

    if (i < k) {
        j = (i + k) / 2 // Find the midpoint in the partition

        // Recursively sort left and right partitions
    }
}
```

```

        MergeSort(numbers, i, j)
        MergeSort(numbers, j + 1, k)

        // Merge left and right partition in sorted order
        Merge(numbers, i, j, k)
    }

main() {
    numbers = { 10, 2, 78, 4, 45, 32, 7, 11 }
    NUMBERS_SIZE = 8
    i = 0

    print("UNSORTED: ")
    for(i = 0; i < NUMBERS_SIZE; ++i) {
        print(numbers[i] + " ")
    }
    printLine()

    MergeSort(numbers, 0, NUMBERS_SIZE - 1)

    print("SORTED: ")
    for(i = 0; i < NUMBERS_SIZE; ++i) {
        print(numbers[i] + " ")
    }
    printLine()
}

```

©zyBooks 06/21/23 22:37 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

UNSORTED: 10 2 78 4 45 32 7 11
 SORTED: 2 4 7 10 11 32 45 78

Merge sort runtime

The merge sort algorithm's runtime is $O(N \log N)$. Merge sort divides the input in half until a list of 1 element is reached, which requires $\log N$ partitioning levels. At each level, the algorithm does about N comparisons selecting and copying elements from the left and right partitions, yielding $N * \log N$ comparisons.

Merge sort requires $O(N)$ additional memory elements for the temporary array of merged elements. For the final merge operation, the temporary list has the same number of elements as the input. Some sorting algorithms sort the list elements in place and require no additional memory, but are more complex to write and understand.

To allocate the temporary array, the `Merge()` function dynamically allocates the array. ©zyBooks 06/21/23 22:37 1692462
Taylor Larrechea
and Summer2023
`mergedNumbers` is a pointer variable that points to the dynamically allocated array, and `new int[mergedSize]` allocates the array with `mergedSize` elements. Alternatively, instead of allocating the array within the `Merge()` function, a temporary array with the same size as the array being sorted can be passed as an argument.



- 1) How many recursive partitioning levels are required for a list of 8 elements?

 //**Check****Show answer**

- 2) How many recursive partitioning levels are required for a list of 2048 elements?

 //**Check****Show answer**

©zyBooks 06/21/23 22:37 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



- 3) How many elements will the temporary merge list have for merging two partitions with 250 elements each?

 //**Check****Show answer****CHALLENGE ACTIVITY****10.3.1: Merge sort.**

489394.3384924.qx3zqy7

Start

numbers:

66	62	42	58	96	65	41	76
----	----	----	----	----	----	----	----

©zyBooks 06/21/23 22:37 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



What call sorts the numbers array?

MergeSort(numbers, Ex: 1 , [])

[Check](#)[Next](#)

©zyBooks 06/21/23 22:37 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

10.4 Bubble sort

Bubble sort is a sorting algorithm that iterates through a list, comparing and swapping adjacent elements if the second element is less than the first element. Bubble sort uses nested loops. Given a list with N elements, the outer i -loop iterates $N - 1$ times. Each iteration moves the largest element into sorted position. The inner j -loop iterates through all adjacent pairs, comparing and swapping adjacent elements as needed, except for the last i pairs that are already in the correct position.

Because of the nested loops, bubble sort has a runtime of $O(N^2)$. Bubble sort is often considered impractical for real-world use because many faster sorting algorithms exist.

Figure 10.4.1: Bubble sort algorithm.

```
BubbleSort(numbers, numbersSize) {
    for (i = 0; i < numbersSize - 1; i++) {
        for (j = 0; j < numbersSize - i - 1;
j++) {
            if (numbers[j] > numbers[j+1]) {
                temp = numbers[j]
                numbers[j] = numbers[j + 1]
                numbers[j + 1] = temp
            }
        }
    }
}
```

©zyBooks 06/21/23 22:37 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

**PARTICIPATION
ACTIVITY**

10.4.1: Bubble sort.



- 1) Bubble sort uses a single loop to sort the list.

 True

- False
- 2) Bubble sort only swaps adjacent elements.
- True
- False
- 3) Bubble sort's best and worst runtime complexity is $O(n^2)$.
- True
- False
- ©zyBooks 06/21/23 22:37 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

10.5 Selection sort



This section has been set as optional by your instructor.

Selection sort

Selection sort is a sorting algorithm that treats the input as two parts, a sorted part and an unsorted part, and repeatedly selects the proper next value to move from the unsorted part to the end of the sorted part.

PARTICIPATION ACTIVITY

10.5.1: Selection sort.



Animation content:

undefined

Animation captions:

©zyBooks 06/21/23 22:37 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

1. Selection sort treats the input as two parts, a sorted and unsorted part. Variables i and j keep track of the two parts.
2. The selection sort algorithm searches the unsorted part of the array for the smallest element; indexSmallest stores the index of the smallest element found.
3. Elements at i and indexSmallest are swapped.
4. Indices for the sorted and unsorted parts are updated.
5. The unsorted part is searched again, swapping the smallest element with the element at i.

6. The process repeats until all elements are sorted.

The index variable i denotes the dividing point. Elements to the left of i are sorted, and elements including and to the right of i are unsorted. All elements in the unsorted part are searched to find the index of the element with the smallest value. The variable `indexSmallest` stores the index of the smallest element in the unsorted part. Once the element with the smallest value is found, that element is swapped with the element at location i . Then, the index i is advanced one place to the right, and the process repeats.

Taylor Larrechea
COLORADOCSPB2270Summer2023

The term "selection" comes from the fact that for each iteration of the outer loop, a value is selected for position i .

PARTICIPATION ACTIVITY

10.5.2: Selection sort algorithm execution.



Assume selection sort's goal is to sort in ascending order.

- 1) Given list (9, 8, 7, 6, 5), what value will be in the 0 element after the first pass over the outer loop ($i = 0$)?

Check

Show answer



- 2) Given list (9, 8, 7, 6, 5), how many swaps will occur during the first pass of the outer loop ($i = 0$)?

Check

Show answer



- 3) Given list (5, 9, 8, 7, 6) and $i = 1$, what will be the list after completing the second outer loop iteration? Type answer as: 1, 2, 3

Check

Show answer



©zyBooks 06/21/23 22:37 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Selection sort runtime

Selection sort has the advantage of being easy to code, involving one loop nested within another loop, as shown below.

Figure 10.5.1: Selection sort algorithm.

```
SelectionSort(numbers, numbersSize) {
    i = 0
    j = 0
    indexSmallest = 0
    temp = 0 // Temporary variable for swap

    for (i = 0; i < numbersSize - 1; ++i) {

        // Find index of smallest remaining element
        indexSmallest = i
        for (j = i + 1; j < numbersSize; ++j) {

            if (numbers[j] < numbers[indexSmallest]) {
                indexSmallest = j
            }
        }

        // Swap numbers[i] and numbers[indexSmallest]
        temp = numbers[i]
        numbers[i] = numbers[indexSmallest]
        numbers[indexSmallest] = temp
    }
}

main() {
    numbers[] = { 10, 2, 78, 4, 45, 32, 7, 11 }
    NUMBERS_SIZE = 8
    i = 0

    print("UNSORTED: ")
    for (i = 0; i < NUMBERS_SIZE; ++i) {
        print(numbers[i] + " ")
    }
    printLine()

    SelectionSort(numbers, NUMBERS_SIZE)

    print("SORTED: ")
    for (i = 0; i < NUMBERS_SIZE; ++i) {
        print(numbers[i] + " ")
    }
    printLine()
}
```

©zyBooks 06/21/23 22:37 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

©zyBooks 06/21/23 22:37 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

UNSORTED: 10 2 78 4 45 32 7 11
SORTED: 2 4 7 10 11 32 45 78

Selection sort may require a large number of comparisons. The selection sort algorithm runtime is $O(N^2)$. If a list has N elements, the outer loop executes $N - 1$ times. For each of those $N - 1$ outer loop executions, the inner loop executes an average of $\frac{N}{2}$ times. So the total number of comparisons is proportional to $\frac{N^2}{2}$, or $O(N^2)$. Other sorting algorithms involve more complex algorithms but have faster execution times.

PARTICIPATION ACTIVITY**10.5.3: Selection sort runtime.**

©zyBooks 06/21/23 22:37 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



Enter an integer value for each answer.

- 1) When sorting a list with 50 elements, `indexSmallest` will be assigned to a minimum of _____ times.

Check**Show answer**

- 2) About how many times longer will sorting a list of $2X$ elements take compared to sorting a list of X elements?

Check**Show answer**

- 3) About how many times longer will sorting a list of $10X$ elements take compared to sorting a list of X elements?

Check**Show answer**

©zyBooks 06/21/23 22:37 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

**CHALLENGE ACTIVITY****10.5.1: Selection sort.**

489394.3384924.qx3zqy7

Start

When using selection sort to sort a list with elements, what is the minimum number of assignments to `indexSmallest` once the outer loop starts?

Ex: 4

1	2	3
---	---	---

©zyBooks 06/21/23 22:37 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Check Next

10.6 Insertion sort



This section has been set as optional by your instructor.

Insertion sort algorithm

Insertion sort is a sorting algorithm that treats the input as two parts, a sorted part and an unsorted part, and repeatedly inserts the next value from the unsorted part into the correct location in the sorted part.

PARTICIPATION ACTIVITY

10.6.1: Insertion sort.



Animation content:

undefined

Animation captions:

1. Variable i is the index of the first unsorted element. Since the element at index 0 is already sorted, i starts at 1.
2. Variable j keeps track of the index of the current element being inserted into the sorted part. If the current element is less than the element to the left, the values are swapped.
3. Once the current element is inserted in the correct location in the sorted part, i is incremented to the next element in the unsorted part.
4. If the current element being inserted is smaller than all elements in the sorted part, that element will be repeatedly swapped with each sorted element until index 0 is reached.
5. Once all elements in the unsorted part are inserted in the sorted part, the list is sorted.

The index variable *i* denotes the starting position of the current element in the unsorted part. Initially, the first element (i.e., element at index 0) is assumed to be sorted, so the outer for loop initializes *i* to 1. The inner while loop inserts the current element into the sorted part by repeatedly swapping the current element with the elements in the sorted part that are larger. Once a smaller or equal element is found in the sorted part, the current element has been inserted in the correct location and the while loop terminates.

©zyBooks 06/21/23 22:37 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Figure 10.6.1: Insertion sort algorithm.

```

InsertionSort(numbers, numbersSize) {
    i = 0
    j = 0
    temp = 0 // Temporary variable for swap

    for (i = 1; i < numbersSize; ++i) {
        j = i
        // Insert numbers[i] into sorted part
        // stopping once numbers[i] in correct position
        while (j > 0 && numbers[j] < numbers[j - 1]) {

            // Swap numbers[j] and numbers[j - 1]
            temp = numbers[j]
            numbers[j] = numbers[j - 1]
            numbers[j - 1] = temp
            --j
        }
    }
}

main() {
    numbers = { 10, 2, 78, 4, 45, 32, 7, 11 }
    NUMBERS_SIZE = 8
    i = 0

    print("UNSORTED: ")
    for(i = 0; i < NUMBERS_SIZE; ++i) {
        print(numbers[i] + " ")
    }
    printLine()

    InsertionSort(numbers, NUMBERS_SIZE)

    print("SORTED: ")
    for(i = 0; i < NUMBERS_SIZE; ++i) {
        print(numbers[i] + " ")
    }
    printLine()
}

```

UNSORTED: 10 2 78 4 45 32 7 11
 SORTED: 2 4 7 10 11 32 45 78

©zyBooks 06/21/23 22:37 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY**10.6.2: Insertion sort algorithm execution.**

Assume insertion sort's goal is to sort in ascending order.

- 1) Given list (20, 14, 85, 3, 9), what value will be in the 0 element after the first pass over the outer loop ($i = 1$)?

 //**Check****Show answer**

©zyBooks 06/21/23 22:37 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 2) Given list (10, 20, 6, 14, 7), what will be the list after completing the second outer loop iteration ($i = 2$)?

Type answer as: 1, 2, 3

 //**Check****Show answer**

- 3) Given list (1, 9, 17, 18, 2), how many swaps will occur during the outer loop execution ($i = 4$)?

 //**Check****Show answer**

Insertion sort runtime

Insertion sort's typical runtime is $O(N^2)$. If a list has N elements, the outer loop executes $N - 1$ times. For each outer loop execution, the inner loop may need to examine all elements in the sorted part. Thus, the inner loop executes on average $\frac{N}{2}$ times. So the total number of comparisons is proportional to $\frac{N(N-1)}{2}$, or $O(N^2)$. Other sorting algorithms involve more complex algorithms but faster execution.

PARTICIPATION ACTIVITY**10.6.3: Insertion sort runtime.**



- 1) In the worst case, assuming each comparison takes 1 μs , how long will insertion sort algorithm take to sort a list of 10 elements?

μs

Check

[Show answer](#)

©zyBooks 06/21/23 22:37 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

- 2) Using the Big O runtime complexity, how many times longer will sorting a list of 20 elements take compared to sorting a list of 10 elements?

//

Check

[Show answer](#)



Nearly sorted lists

For sorted or nearly sorted inputs, insertion sort's runtime is $O(N)$. A **nearly sorted** list only contains a few elements not in sorted order. Ex: (4, 5, 17, 25, 89, 14) is nearly sorted having only one element not in sorted position.

PARTICIPATION ACTIVITY

10.6.4: Nearly sorted lists.



Determine if each of the following lists is unsorted, sorted, or nearly sorted. Assume ascending order.

- 1) (6, 14, 85, 102, 102, 151)



- Unsorted
- Sorted
- Nearly sorted

©zyBooks 06/21/23 22:37 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

- 2) (23, 24, 36, 48, 19, 50, 101)

- Unsorted
- Sorted
- Nearly sorted

- 3) (15, 19, 21, 24, 2, 3, 6, 11)



- Unsorted
- Sorted
- Nearly sorted

©zyBooks 06/21/23 22:37 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Insertion sort runtime for nearly sorted input

For each outer loop execution, if the element is already in sorted position, only a single comparison is made. Each element not in sorted position requires at most N comparisons. If there are a constant number, C, of unsorted elements, sorting the N - C sorted elements requires one comparison each, and sorting the C unsorted elements requires at most N comparisons each. The runtime for nearly sorted inputs is $O((N - C) * 1 + C * N) = O(N)$.

PARTICIPATION ACTIVITY

10.6.5: Using insertion sort for nearly sorted list.



Animation captions:

1. Sorted part initially contains the first element.
2. An element already in sorted position only requires a single comparison, which is O(1) complexity.
3. An element not in sorted position requires O(N) comparisons. For nearly sorted inputs, insertion sort's runtime is O(N).

PARTICIPATION ACTIVITY

10.6.6: Insertion sort algorithm execution for nearly sorted input.



Assume insertion sort's goal is to sort in ascending order.

- 1) Given list (10, 11, 12, 13, 14, 15), how many comparisons will be made during the third outer loop execution ($i = 3$)?

Check**Show answer**©zyBooks 06/21/23 22:37 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 2) Given list (10, 11, 12, 13, 14, 7), how many comparisons will be made



during the final outer loop execution ($i = 5$)?

 //**Check****Show answer**

©zyBooks 06/21/23 22:37 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

- 3) Given list (18, 23, 34, 75, 3), how many total comparisons will insertion sort require?

 //**Check****Show answer****CHALLENGE ACTIVITY**

10.6.1: Insertion sort.



489394.3384924.qx3zqy7

Start

Given list (20, 25, 26, 35, 37, 39, 24, 38, 30), what is the value of i when the first swap executes?

Ex: 1

1	2	3	4	5
---	---	---	---	---

Check**Next**

©zyBooks 06/21/23 22:37 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

10.7 Shell sort



This section has been set as optional by your instructor.

Shell sort's interleaved lists

Shell sort is a sorting algorithm that treats the input as a collection of interleaved lists, and sorts each list individually with a variant of the insertion sort algorithm. Shell sort uses gap values to determine the number of interleaved lists. A **gap value** is a positive integer representing the distance between elements in an interleaved list. For each interleaved list, if an element is at index i , the next element is at index $i + \text{gap value}$.

Shell sort begins by choosing a gap value K and sorting K interleaved lists in place. Shell sort finishes by performing a standard insertion sort on the entire array. Because the interleaved parts have already been sorted, smaller elements will be close to the array's beginning and larger elements towards the end. Insertion sort can then quickly sort the nearly-sorted array.

Any positive integer gap value can be chosen. In the case that the array size is not evenly divisible by the gap value, some interleaved lists will have fewer items than others.

PARTICIPATION ACTIVITY

10.7.1: Sorting interleaved lists with shell sort speeds up insertion sort.



Animation captions:

1. If a gap value of 3 is chosen, shell sort views the list as 3 interleaved lists. 56, 12, and 75 make up the first list, 42, 77, and 91 the second, and 93, 82, and 36 the third.
2. Shell sort will sort each of the 3 lists with insertion sort.
3. The result is not a sorted list, but is closer to sorted than the original. Ex: The 3 smallest elements, 12, 42, and 36, are the first 3 elements in the list.
4. Sorting the original array with insertion sort requires 17 swaps.
5. Sorting the interleaved lists required 4 swaps. Running insertion sort on the array requires 7 swaps total, far fewer than insertion sort on the original array.

PARTICIPATION ACTIVITY

10.7.2: Shell sort's interleaved lists.



For each question, assume a list with 6 elements.

©zyBooks 06/21/23 22:37 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



- 1) With a gap value of 3, how many interleaved lists will be sorted?

- 1
- 2
- 3

6

- 2) With a gap value of 3, how many items will be in each interleaved list?

 1 2 3 6

©zyBooks 06/21/23 22:37 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



- 3) If a gap value of 2 is chosen, how many interleaved lists will be sorted?

 1 2 3 6

- 4) If a gap value of 4 is chosen, how many interleaved lists will be sorted?

A gap value of 4 cannot be used on an array with 6 elements.

 2 3 4

Insertion sort for interleaved lists

If a gap value of K is chosen, creating K entirely new lists would be computationally expensive. Instead of creating new lists, shell sort sorts interleaved lists in-place with a variation of the insertion sort algorithm. The insertion sort algorithm variant redefines the concept of "next" and "previous" items. For an item at index X, the next item is at $X + K$, instead of $X + 1$, and the previous item is at $X - K$ instead of $X - 1$.

PARTICIPATION ACTIVITY

10.7.3: Interleaved insertion sort.

©zyBooks 06/21/23 22:37 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Animation content:

undefined

Animation captions:

1. Calling `InsertionSortInterleaved` with a start index of 0 and a gap of 3 sorts the interleaved list with elements at indices 0, 3, and 6. i and j are first assigned with index 3.
2. When swapping the 2 elements 45 and 88, 45 jumps the gap and moves towards the front more quickly compared to the regular insertion sort.
3. The sort continues, putting 45, 71, and 88 in the correct order.
4. Only 1 of 3 interleaved lists has been sorted. 2 more `InsertionSortInterleaved` calls are needed, with a start index of 1 for the second list, and 2 for the third list.
5. Calling `InsertionSortInterleaved` with a starting index of 0 and a gap of 1 is equivalent to the regular insertion sort, and finishes sorting the list.

©zyBooks 06/21/23 22:37 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

10.7.4: Insertion sort variant.



For each call to `InsertionSortInterleaved(numbersArray, x, ...)`, assume that `numbersArray` is an array of length X. Each question can be answered without knowing the contents of `numbersArray`.

- 1) `InsertionSortInterleaved(numbersArray, 10, 0, 5)` is called. What are the indices of the first two elements compared?

- 1 and 5
- 1 and 6
- 0 and 4
- 0 and 5

- 2) `InsertionSortInterleaved(numbersArray, 4, 1, 4)` is called. What is the value of the loop variable i first initialized to?

- 1
- 4
- 5

- 3) `InsertionSortInterleaved(numbersArray, 4, 1, 4)` results in an out of bounds array access.

©zyBooks 06/21/23 22:37 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- True
- False

- 4) If a gap value of 2 is chosen, then the following two function calls will fully sort `numbersArray`:



```
InsertionSortInterleaved(numbersArray,  
9, 0, 2)  
InsertionSortInterleaved(numbersArray,  
9, 1, 2)
```

- True
- False

©zyBooks 06/21/23 22:37 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Shell sort algorithm

Shell sort begins by picking an arbitrary collection of gap values. For each gap value K, K calls are made to the insertion sort variant function to sort K interleaved lists. Shell sort ends with a final gap value of 1, to finish with the regular insertion sort.

Shell sort tends to perform well when choosing gap values in descending order. A common option is to choose powers of 2 minus 1, in descending order. Ex: For an array of size 100, gap values would be 63, 31, 15, 7, 3, and 1. This gap selection technique results in shell sort's time complexity being no worse than .

Using gap values that are powers of 2 or in descending order is not required. Shell sort will correctly sort arrays using any positive integer gap values in any order, provided a gap value of 1 is included.

PARTICIPATION ACTIVITY

10.7.5: Shell sort algorithm.



Animation content:

undefined

Animation captions:

1. The first gap value of 5 causes 5 interleaved lists to be sorted. The inner for loop iterates over the start indices for the interleaved list. So, i is initialized with the first list's starting index, or 0.
2. The second for loop iteration sorts the interleaved list starting at index 1 with the same gap value of 5.
3. For the gap value 5, the remaining interleaved lists at start indices 2, 3, and 4 are sorted.
4. The next gap value of 3 causes 3 interleaved lists to be sorted. Few swaps are needed because the list is already partially sorted.
5. The final gap value of 1 finishes sorting.

©zyBooks 06/21/23 22:37 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

10.7.6: ShellSort.



1) ShellSort will properly sort an array using any collection of gap values, provided the collection contains 1.

- True
- False

©zyBooks 06/21/23 22:37 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

2) Calling ShellSort with gap array (7, 3, 1) vs. (3, 7, 1) produces the same result with no difference in efficiency.

- True
- False



3) How many times is

InsertionSortInterleaved called if
ShellSort is called with gap array (10, 2,
1)?

- 3
- 12
- 13
- 20

**CHALLENGE ACTIVITY**

10.7.1: Shell sort.



489394.3384924.qx3zqy7

Start

Given an array [80, 88, 40, 55, 94, 24, 74, 54, 81] and a gap value of 4:

What is the first interleaved array?

[Ex: 1, 2, 3]
(comma between values)

©zyBooks 06/21/23 22:37 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

What is the second interleaved array?

[]

[Check](#)[Next](#)

10.8 Radix sort

©zyBooks 06/21/23 22:37 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



This section has been set as optional by your instructor.

Buckets

Radix sort is a sorting algorithm designed specifically for integers. The algorithm makes use of a concept called buckets and is a type of bucket sort.

Any array of integer values can be subdivided into buckets by using the integer values' digits. A **bucket** is a collection of integer values that all share a particular digit value. Ex: Values 57, 97, 77, and 17 all have a 7 as the 1's digit, and would all be placed into bucket 7 when subdividing by the 1's digit.

PARTICIPATION ACTIVITY

10.8.1: A particular single digit in an integer can determine the integer's bucket.



Animation captions:

1. Using only the 1's digit, each integer can be put into a bucket. 736 is put into bucket 6, 81 into bucket 1, 101 into bucket 1, and so on.
2. Using only the 10's digit, each integer can be put into a bucket. 736 is put into bucket 3, 81 into bucket 8, and so on. 5 is like 05 so is put into bucket 0.
3. Using only the 100's digit, each integer can be put into a bucket. 736 is put into bucket 7, 81 is like 081 so is put into bucket 0, and so on.

PARTICIPATION ACTIVITY

10.8.2: Using the 1's digit, place each integer in the correct bucket.



If unable to drag and drop, refresh the page.

©zyBooks 06/21/23 22:37 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

49**74****7****50****Bucket 0**

Bucket 4

Bucket 7

Bucket 9

©zyBooks 06/21/23 22:37 1692462

Reset

Taylor Larrechea

COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

10.8.3: Using the 10's digit, place each integer in the correct bucket.



If unable to drag and drop, refresh the page.

7 **86** **50** **74**

Bucket 0

Bucket 5

Bucket 7

Bucket 8

Reset**PARTICIPATION ACTIVITY**

10.8.4: Bucket concepts.



- 1) Integers will be placed into buckets based on the 1's digit. More buckets are needed for an array with one thousand integers than for an array with one hundred integers.

- True
 False

- 2) Consider integers X and Y, such that X < Y. X will always be in a lower bucket than Y.

©zyBooks 06/21/23 22:37 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- True
- False

3) All integers from an array could be placed into the same bucket, even if the array has no duplicates.

- True
- False

©zyBooks 06/21/23 22:37 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



PARTICIPATION ACTIVITY

10.8.5: Assigning integers to buckets.



For each question, consider the array of integers: 51, 47, 96, 52, 27.

1) When placing integers using the 1's digit, how many integers will be in bucket 7?

- 0
- 1
- 2



2) When placing integers using the 1's digit, how many integers will be in bucket 5?

- 0
- 1
- 2



3) When placing integers using the 10's digit, how many will be in bucket 9?

- 0
- 1
- 2

©zyBooks 06/21/23 22:37 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



4) All integers would be in bucket 0 if using the 100's digit.

- True
- False



Radix sort algorithm

Radix sort is a sorting algorithm specifically for an array of *integers*: The algorithm processes one digit at a time starting with the least significant digit and ending with the most significant. Two steps are needed for each digit. First, all array elements are placed into buckets based on the current digit's value. Then, the array is rebuilt by removing all elements from buckets, in order from lowest bucket to highest.

©zyBooks 06/21/23 22:37 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

PARTICIPATION
ACTIVITY

10.8.6: Radix sort algorithm (for non-negative integers).

Animation content:

undefined

Animation captions:

1. Radix sort begins by allocating 10 buckets and putting each number in a bucket based on the 1's digit.
2. Numbers are taken out of buckets, in order from lowest bucket to highest, rebuilding the array.
3. The process is repeated for the 10's digit. First, the array numbers are placed into buckets based on the 10's digit.
4. The items are copied from buckets back into the array. Since all digits have been processed, the result is a sorted array.

Figure 10.8.1: RadixGetMaxLength and RadixGetLength functions.

©zyBooks 06/21/23 22:37 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

```
// Returns the maximum length, in number of digits, out of all elements in
// the array
RadixGetMaxLength(array, arraySize) {
    maxDigits = 0
    for (i = 0; i < arraySize; i++) {
        digitCount = RadixGetLength(array[i])
        if (digitCount > maxDigits)
            maxDigits = digitCount
    }
    return maxDigits
}

// Returns the length, in number of digits, of value
RadixGetLength(value) {
    if (value == 0)
        return 1

    digits = 0
    while (value != 0) {
        digits = digits + 1
        value = value / 10
    }
    return digits
}
```

©zyBooks 06/21/23 22:37 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

**PARTICIPATION
ACTIVITY**

10.8.7: Radix sort algorithm.



- 1) What will RadixGetLength(17)
evaluate to?

 //**Check****Show answer**

- 2) What will RadixGetMaxLength
return when the array is (17, 4,
101)?

 //

©zyBooks 06/21/23 22:37 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



- 3) When sorting the array (57, 5, 501)
with RadixSort, what is the largest
number of integers that will be in
bucket 5 at any given moment?



**Check****Show answer**

©zyBooks 06/21/23 22:37 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

10.8.8: Radix sort algorithm analysis.



- 1) When sorting an array of n 3-digit integers, RadixSort's worst-case time complexity is $O(n)$.

- True
- False



- 2) When sorting an array with n elements, the maximum number of elements that RadixSort may put in a bucket is n .

- True
- False



- 3) RadixSort has a space complexity of $O(1)$.

- True
- False



- 4) The RadixSort() function shown above also works on an array of floating-point values.

- True
- False



©zyBooks 06/21/23 22:37 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Sorting signed integers

The above radix sort algorithm correctly sorts arrays of non-negative integers. But if the array contains negative integers, the above algorithm would sort by absolute value, so the integers are not correctly sorted. A small extension to the algorithm correctly handles negative integers.

In the extension, before radix sort completes, the algorithm allocates two buckets, one for negative integers and the other for non-negative integers. The algorithm iterates through the array in order,

placing negative integers in the negative bucket and non-negative integers in the non-negative bucket. The algorithm then reverses the order of the negative bucket and concatenates the buckets to yield a sorted array. Pseudocode for the completed radix sort algorithm follows.

Figure 10.8.2: RadixSort algorithm (for negative and non-negative integers).

©zyBooks 06/21/23 22:37 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

```
RadixSort(array, arraySize) {
    buckets = create array of 10 buckets

    // Find the max length, in number of digits
    maxDigits = RadixGetMaxLength(array, arraySize)

    pow10 = 1
    for (digitIndex = 0; digitIndex < maxDigits;
    digitIndex++) {
        for (i = 0; i < arraySize; i++) {
            bucketIndex = abs(array[i] / pow10) % 10
            Append array[i] to buckets[bucketIndex]
        }
        arrayIndex = 0
        for (i = 0; i < 10; i++) {
            for (j = 0; j < buckets[i]→size(); j++) {
                array[arrayIndex] = buckets[i][j]
                arrayIndex = arrayIndex + 1
            }
        }
        pow10 = pow10 * 10
        Clear all buckets
    }

    negatives = all negative values from array
    nonNegatives = all non-negative values from array
    Reverse order of negatives
    Concatenate negatives and nonNegatives into array
}
```

PARTICIPATION ACTIVITY

10.8.9: Sorting signed integers.



©zyBooks 06/21/23 22:37 1692462

COLORADOCSPB2270Summer2023

For each question, assume radix sort has sorted integers by absolute value to produce the array (-12, 23, -42, 73, -78), and is about to build the negative and non-negative buckets to complete the sort.

- 1) What integers will be placed into the negative bucket? Type answer as: 15, 42, 98



//**Check****Show answer**

- 2) What integers will be placed into the non-negative bucket? Type answer as: 15, 42, 98

 //**Check****Show answer**

©zyBooks 06/21/23 22:37 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 3) After reversal, what integers are in the negative bucket? Type answer as: 15, 42, 98

 //**Check****Show answer**

- 4) What is the final array after RadixSort concatenates the two buckets? Type answer as: 15, 42, 98

 //**Check****Show answer**

Radix sort with different bases

This section presents radix sort with base 10, but other bases can be used as well. Ex: Using base 2 is another common approach, where only 2 buckets would be required, instead of 10.

**CHALLENGE
ACTIVITY****10.8.1: Radix sort.**

489394.3384924.qx3zqy7

Start

Using only the 1's digit,

what is the correct bucket for 96?

Ex: 5

what is the correct bucket for 408?

Using only the 10's digit,

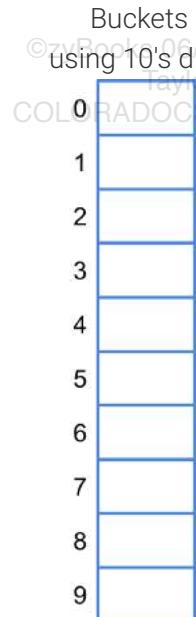
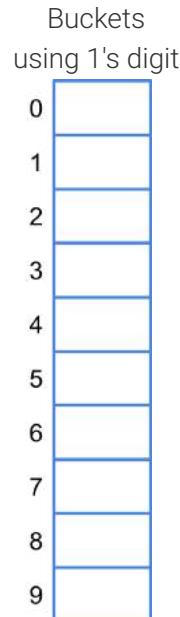
what is the correct bucket for 96?

what is the correct bucket for 408?

Using only the 100's digit,

what is the correct bucket for 96?

what is the correct bucket for 408?

**Check****Next**

10.9 Overview of fast sorting algorithms



This section has been set as optional by your instructor.

Fast sorting algorithm

A **fast sorting algorithm** is a sorting algorithm that has an average runtime complexity $O(n \log n)$ or better. The table below shows average runtime complexities for several sorting algorithms.

Table 10.9.1: Sorting algorithms' average runtime complexity.

Sorting algorithm	Average case runtime complexity	Fast?
Selection sort	$O(n^2)$	No
Insertion sort	$O(n^2)$	No
Shell sort	$O(n^2)$	No
Quicksort	$O(n \log n)$	Yes
Merge sort	$O(n \log n)$	Yes
Heap sort	$O(n \log n)$	Yes
Radix sort	$O(n)$	Yes

PARTICIPATION ACTIVITY**10.9.1: Fast sorting algorithms.**

1) Insertion sort is a fast sorting algorithm.



- True
- False

2) Merge sort is a fast sorting algorithm.



- True
- False

3) Radix sort is a fast sorting algorithm.



- True
- False

Comparison sorting

A **element comparison sorting algorithm** is a sorting algorithm that operates on an array of elements that can be compared to each other. Ex: An array of strings can be sorted with a comparison sorting algorithm, since two strings can be compared to determine if the one string is less than, equal to, or greater than another string. Selection sort, insertion sort, shell sort, quicksort, merge sort, and heap

©zyBooks 06/21/23 22:37 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

sort are all comparison sorting algorithms. Radix sort, in contrast, subdivides each array element into integer digits and is not a comparison sorting algorithm.

Table 10.9.2: Identifying comparison sorting algorithms.

Sorting algorithm	Comparison?
Selection sort	Yes
Insertion sort	Yes
Shell sort	Yes
Quicksort	Yes
Merge sort	Yes
Heap sort	Yes
Radix sort	No

©zyBooks 06/21/23 22:37 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

10.9.2: Comparison sorting algorithms.



- 1) Selection sort can be used to sort an array of strings.



- True
- False

- 2) The fastest average runtime complexity of a comparison sorting algorithm is $O(\quad)$.



- True
- False

©zyBooks 06/21/23 22:37 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Best and worst case runtime complexity

A fast sorting algorithm's best or worst case runtime complexity may differ from the average runtime complexity. Ex: The best and average case runtime complexity for quicksort is $O(\quad)$, but the worst case is $O(\quad)$.

Table 10.9.3: Fast sorting algorithm's best, average, and worst case runtime complexity.

Sorting algorithm	Best case runtime complexity	Average case runtime complexity	Worst case runtime complexity
Quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Heap sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Radix sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

PARTICIPATION ACTIVITY

10.9.3: Runtime complexity.



- 1) A fast sorting algorithm's worst case runtime complexity must be $O(n^2)$ or better.



- True
- False

- 2) Which fast sorting algorithm's worst case runtime complexity is worse than $O(n \log n)$?



- Quicksort
- Heap sort
- Radix sort

11.1 B-trees

Introduction to B-trees

In a binary tree, each node has one key and up to two children. A **B-tree** with order K is a tree where nodes can have up to K-1 keys and up to K children. The **order** is the maximum number of children a node can have. Ex: In a B-tree with order 4, a node can have 1, 2, or 3 keys, and up to 4 children. B-trees have the following properties:

- All keys in a B-tree must be distinct.
- All leaf nodes must be at the same level.
- An internal node with N keys must have N+1 children.
- Keys in a node are stored in sorted order from smallest to largest.
- Each key in a B-tree internal node has one left subtree and one right subtree. All left subtree keys are < that key, and all right subtree keys are > that key.

PARTICIPATION ACTIVITY

11.1.1: Order 3 B-trees.



Animation captions:

1. A single node in a B-tree can contain multiple keys.
2. An order 3 B-tree can have up to 2 keys per node. This root node contains the keys 10 and 20, which are ordered from smallest to largest.
3. An internal node with 2 keys must have three children. The node with keys 10 and 20 has three children nodes, with keys 5, 15, and 25.
4. The root's left subtree contains the key 5, which is less than 10.
5. The root's middle subtree contains the key 15, which is greater than 10 and less than 20.
6. The root's right subtree contains the key 25, which is greater than 20.
7. All left subtree keys are < the parent key, and all right subtree keys are > the parent key.

PARTICIPATION ACTIVITY

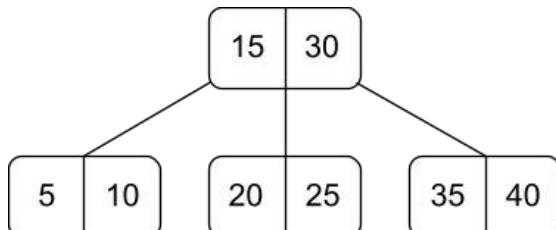
11.1.2: Validity of order 3 B-trees.



Determine which of the following are valid order 3 B-trees.

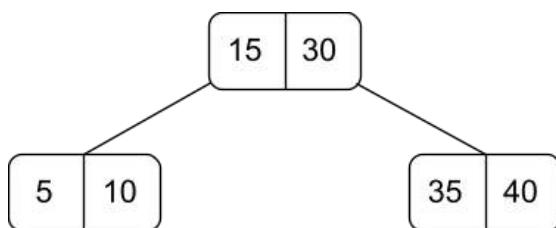
©zyBooks 06/21/23 23:08 169246
Taylor Larrechea
COLORADOCSPB2270Summer2023

1)



Valid Invalid

2)

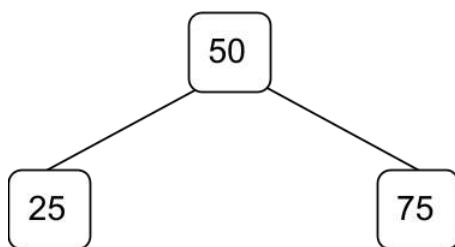
 Valid Invalid

©zyBooks 06/21/23 23:08 1692462

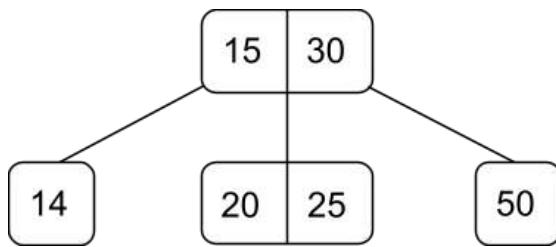
Taylor Larrechea

COLORADOCSPB2270Summer2023

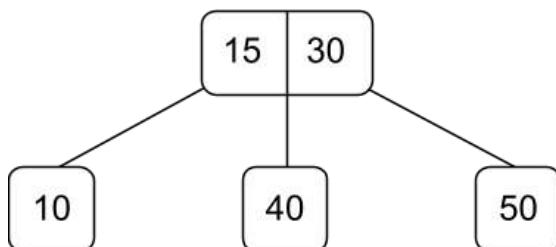
3)

 Valid Invalid

4)

 Valid Invalid

5)

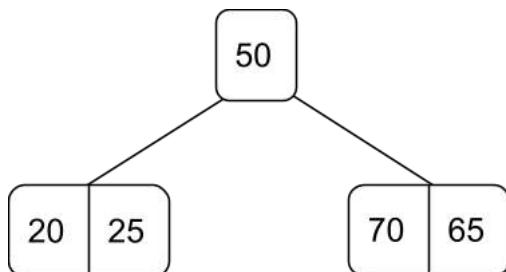
 Valid Invalid

©zyBooks 06/21/23 23:08 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

6)



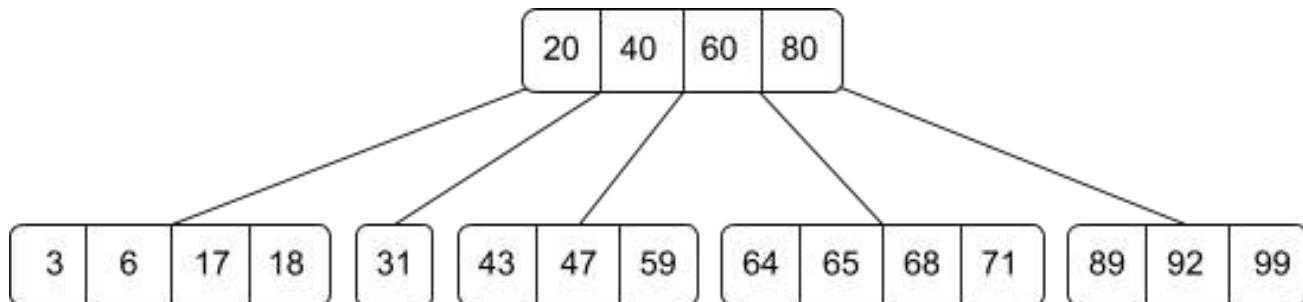
- Valid
- Invalid

©zyBooks 06/21/23 23:08 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Higher order B-trees

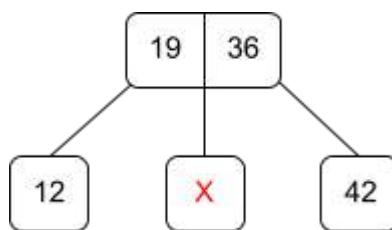
As the order of a B-trees increases, the maximum number of keys and children per node increases. An internal node must have one more child than keys. Each child of an internal node can have a different number of keys than the parent internal node. Ex: An internal node in an order 5 B-tree could have 1 child with 1 key, 2 children with 3 keys, and 2 children with 4 keys.

Example 11.1.1: A valid order 5 B-tree.



PARTICIPATION ACTIVITY

11.1.3: B-tree properties.



©zyBooks 06/21/23 23:08 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- What is the minimum possible order of this B-tree?

Check

Show answer

- 2) What is the minimum possible integer value for the unknown key X?

Check**Show answer**

©zyBooks 06/21/23 23:08 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



- 3) What is the maximum possible integer value for the unknown key X?

Check**Show answer**

2-3-4 Trees

A 2-3-4 tree is an order 4 B-tree. Therefore, a 2-3-4 tree node contains 1, 2 or 3 keys. A leaf node in a 2-3-4 tree has no children.

Table 11.1.1: 2-3-4 tree internal nodes.

Number of keys	Number of children
1	2
2	3
3	4

PARTICIPATION ACTIVITY

11.1.4: 2-3-4 tree properties.

©zyBooks 06/21/23 23:08 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



- 1) A 2-3-4 tree is a B-tree of order



Check**Show answer**

- 2) What is the minimum number of children that a 2-3-4 internal node with 2 keys can have?

//

Check**Show answer**

©zyBooks 06/21/23 23:08 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



- 3) What is the maximum number of children that a 2-3-4 internal node with 2 keys can have?

//

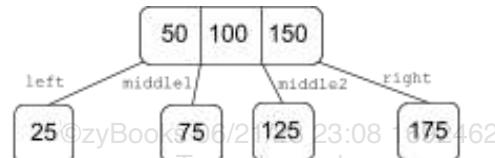
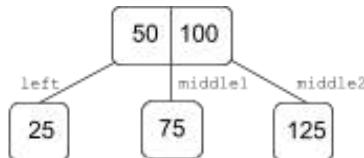
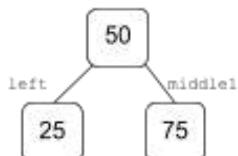
Check**Show answer**

2-3-4 tree node labels

The keys in a 2-3-4 tree node are labeled as A, B and C. The child nodes of a 2-3-4 tree internal node are labeled as left, middle1, middle2, and right. If a node contains 1 key, then keys B and C, as well as children middle2 and right, are not used. If a node contains 2 keys, then key C, as well as the right child, are not used. A 2-3-4 tree node containing exactly 3 keys is said to be **full**, and uses all keys and children.

A node with 1 key is called a **2-node**. A node with 2 keys is called a **3-node**. A node with 3 keys is called a **4-node**.

Figure 11.1.1: 2-3-4 child subtree labels.

©zyBooks 06/21/23 23:08 1692462
Taylor Larrechea

COLORADOCSPB2270Summer2023


PARTICIPATION ACTIVITY

11.1.5: 2-3-4 tree nodes.

- 1) Every 2-3-4 tree internal node will have children left and _____.



//**Show answer**

- 2) The right child is only used by a 2-3-4 tree internal node with _____ keys.

 //**Check****Show answer**

©zyBooks 06/21/23 23:08 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

- 3) A node in a 2-3-4 tree that contains no children is called a _____ node.

 //**Check****Show answer**

- 4) A 2-3-4 tree node with _____ keys is said to be full.

 //**Check****Show answer**

11.2 2-3-4 tree search algorithm

Given a key, a **search** algorithm returns the first node found matching that key, or returns null if a matching node is not found. Searching a 2-3-4 tree is a recursive process that starts with the root node. If the search key equals any of the keys in the node, then the node is returned. Otherwise, a recursive call is made on the appropriate child node. Which child node is used depends on the value of the search key in comparison to the node's keys. The table below shows conditions, which are checked in order, and the corresponding child nodes.

Table 11.2.1: 2-3-4 tree child node to choose based on search key.

Condition	Child node to search
key < node's A key	left
node has only 1 key or key < node's B key	middle1 ©zyBooks 06/21/23 23:08 1692462 Taylor Larrechea
node has only 2 keys or key < node's C key	middle2 COLORADOCSPB2270Summer2023
none of the above	right

PARTICIPATION ACTIVITY

11.2.1: 2-3-4 tree search algorithm.

**Animation content:**

Static figure: A code block and a 2-3-4 tree labeled BTTreeSearch(tree → root, 70).

Begin pseudocode:

```
BTTreeSearch(node, key) {
    if (node is not null) {
        if (node has key) {
            return node
        }
        if (key < node → A) {
            return BTTreeSearch(node → left, key)
        }
        else if (node → B is null || key < node → B) {
            return BTTreeSearch(node → middle1, key)
        }
        else if (node → C is null || key < node → C) {
            return BTTreeSearch(node → middle2, key)
        }
        else {
            return BTTreeSearch(node → right, key)
        }
    }
    return null
}
```

End pseudocode.

©zyBooks 06/21/23 23:08 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Static figure: Code for BTreesearch() function is shown on the left. On the right is a 2-3-4 tree with four nodes and six keys. Root's keys are 25 and 50. Root's left child has key 10. Root's middle1 child has keys 35 and 40. Root's middle2 child has key 70. Above the tree is the search function call: BTreesearch(tree \rightarrow root, 70). A label with text "node" has an accompanying arrow that points to node 70.

Step 1: Execution of BTreesearch(tree \rightarrow root, 70) begins. Execution highlight is shown on function signature, not yet executing code in the function body. A label "node" appears with an accompanying arrow pointing to the root node.

©zyBooks 06/21/23 23:08 1692462
May 2023
COLORADOCSPB2270Summer2023

Step 2: The first if statement executes and the condition is true since node is not null. Code highlight proceeds to the next if statement's condition, "node has key". The comparisons that occur are shown: $70 \neq 25$ and $70 \neq 50$. Neither is true, so the code highlight proceeds to the closing curly brace for the if statement.

Step 3: The next if statement's condition is evaluated and is false because 70 is not less than 25. The next else-if statement's condition is also false because 70 is not less than 50. The following else-if statement's condition is true because node \rightarrow C is null.

Step 4: The following recursive call executes: BTreesearch(node \rightarrow middle2, key). Code execution moves back to the function's top line and the node label and arrow move to node 70.

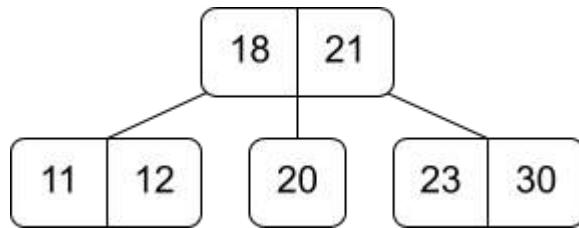
Step 5: The first two if statement's condition is evaluated and is true since node is not null. The next if statement's condition is also true since $70 == 70$, meaning the node has the key. So execution reaches the "return node" statement and the search is done.

Animation captions:

1. Search for 70 starts at the root node.
2. node is not null, so the search compares 70 with the node's two keys, 25 and 50. No match occurs, so the node does not have key 70.
3. Since no match was found in the root node, the search algorithm compares the key to the node's keys to determine the recursive call.
4. 70 is greater than 50, and the node does not contain a key C, so a recursive call to the middle2 child node occurs.
5. node is not null, so 70 is compared with the node's A key. A match occurs, so the node is returned.

©zyBooks 06/21/23 23:08 1692462
COLORADOCSPB2270Summer2023





- 1) When searching for key 23, what node is visited first?

- Root
- Root's left child
- Root's middle1 child
- Root's middle2 child

- 2) When searching for key 23, how many keys in the root are compared against 23?

- 1
- 2
- 3
- 4

- 3) When searching for key 23, the root node will be the only node that is visited.

- True
- False

- 4) When searching for key 23, what is the total number of nodes visited?

- 1
- 2
- 3
- 4

- 5) When searching for key 20, what is returned by the search function?

- Null
- Root's left child
- Root's middle1 child

©zyBooks 06/21/23 23:08 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

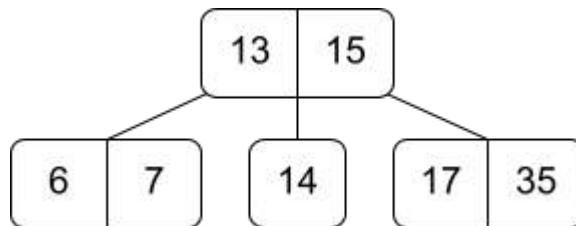
- Root's middle2 child
- 6) When searching for key 19, what is returned by the search function?

- Null
- Root's left child
- Root's middle1 child
- Root's middle2 child

©zyBooks 06/21/23 23:08 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

11.2.3: 2-3-4 tree search algorithm.



- 1) When searching for key 6, search starts at the root. Since the root node does not contain the key 6, which recursive search call is made?

- BTreeSearch(node->left, key)
- BTreeSearch(node->middle1, key)
- BTreeSearch(node->middle2, key)
- BTreeSearch(node->right, key)

- 2) When searching for key 6, after making the recursive call on the root's left node, which return statement is executed?

- return BTreeSearch(node->left, key)
- return node->A
- return node
- return null

©zyBooks 06/21/23 23:08 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



3) When searching for key 15, which recursive search call is made?

- BTreeSearch(node->left, key)
- BTreeSearch(node->middle1, key)
- no recursive call is made

©zyBooks 06/21/23 23:08 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

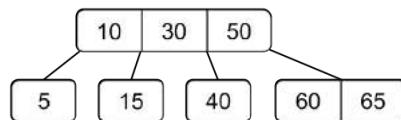
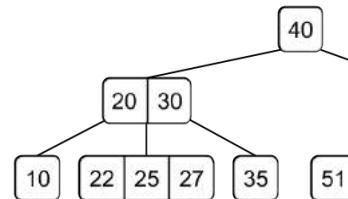
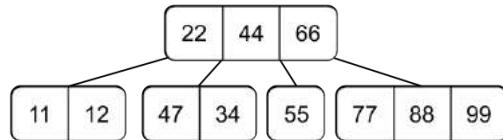
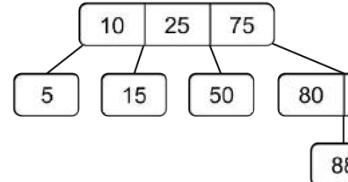
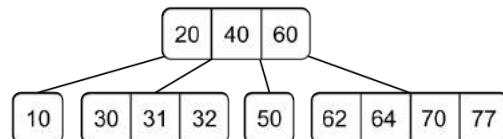
CHALLENGE ACTIVITY

11.2.1: 2-3-4 tree search algorithm.



489394.3384924.qx3zqy7

Select all valid 2-3-4 trees.

 A

 D

 B

 E

 C


©zyBooks 06/21/23 23:08 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

1

2

3

4

11.3 2-3-4 tree insert algorithm

2-3-4 tree insertions and split operations

Given a new key, a 2-3-4 tree **insert** operation inserts the new key in the proper location such that all 2-3-4 tree properties are preserved. New keys are always inserted into leaf nodes in a 2-3-4 tree.

An important operation during insertion is the **split** operation, which is done on every full node encountered during insertion traversal. The split operation moves the middle key from a child node into the child's parent node. The first and last keys in the child node are moved into two separate nodes. The split operation returns the parent node that received the middle key from the child.

PARTICIPATION ACTIVITY

11.3.1: Split operation.



Animation captions:

1. To split the full root node, the middle key moves up, becoming the new root node with a single value.
2. To split a full, non-root node, the middle value is moved up into the parent node.
3. Compared to the original, the tree contains the same values after the split, and all 2-3-4 tree requirements are still satisfied.

PARTICIPATION ACTIVITY

11.3.2: Split operation.



- 1) During insertion, only a full node can be split.
 - True
 - False
- 2) During insertion of a key K, after splitting a node, the key K is immediately inserted into the node.
 - True
 - False
- 3) What is the result of splitting a full root node?

©zyBooks 06/21/23 23:08 169246
Taylor Larrechea
COLORADOCSPB2270Summer2023

©zyBooks 06/21/23 23:08 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- The total number of nodes in the tree decreases by 1.
 - The total number of nodes in the tree does not change.
 - The total number of nodes in the tree increases by 1.
 - The total number of nodes in the tree increases by 2.
- 4) When a full internal node is split, which key moves up into the parent node?
- First
 - Middle
 - Last

Split operation algorithm

Splitting an internal node allocates 2 new nodes, each with a single key, and the middle key from the split node moves up into the parent node. Splitting the root node allocates 3 new nodes, each with a single key, and the root of the tree becomes a new node with a single key.

PARTICIPATION ACTIVITY

11.3.3: B-tree split operation.



Animation content:

undefined

©zyBooks 06/21/23 23:08 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Animation captions:

1. Splitting a node starts by verifying that the node is full. A pointer to the parent node is also needed when splitting an internal node.

2. New node allocation is necessary. splitLeft is allocated with a single key copied from node \rightarrow A, and two null child pointers copied from node \rightarrow left and node \rightarrow middle1.
3. splitRight is allocated with a single key copied from node \rightarrow C, and null child pointers copied from node \rightarrow middle2 and node \rightarrow right.
4. Since nodeParent is not null, the key 37 moves from node into nodeParent and the two newly allocated children are attached to nodeParent as well.
5. Splitting the root node allocates 3 new nodes, one of which becomes the new root.

©zyBooks 06/21/23 23:08 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

During a split operation, any non-full internal node may need to gain a key from a split child node. This key may have children on either side.

Figure 11.3.1: Inserting a key with children into a non-full parent node.

```
BTreeInsertKeyWithChildren(parent, key, leftChild,
rightChild) {
    if (key < parent->A) {
        parent->C = parent->B
        parent->B = parent->A
        parent->A = key
        parent->right = parent->middle2
        parent->middle2 = parent->middle1
        parent->middle1 = rightChild
        parent->left = leftChild
    }
    else if (parent->B is null || key < parent->B) {
        parent->C = parent->B
        parent->B = key
        parent->right = parent->middle2
        parent->middle2 = rightChild
        parent->middle1 = leftChild
    }
    else {
        parent->C = key
        parent->right = rightChild
        parent->middle2 = leftChild
    }
}
```

PARTICIPATION ACTIVITY

11.3.4: B-tree split operation.

©zyBooks 06/21/23 23:08 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 1) Like searching, the split operation in a 2-3-4 tree is recursive.

- True
- False



- 2) If a non-full node is passed to BTreeSplit, then the root node is returned.

- True
- False

- 3) Allocating new nodes is necessary for the split operation.

©zyBooks 06/21/23 23:08 169246
Taylor Larrechea
COLORADOCSPB2270Summer2023

- True
- False

- 4) The root node is split in the same way a non-root node is split.

- True
- False

- 5) When splitting a node, a pointer to the node's parent is required.

- True
- False

- 6) The split function should always split a node, even if the node is not full.

- True
- False

Inserting a key into a leaf node

A new key is always inserted into a non-full leaf node. The table below describes the 4 possible cases for inserting a new key into a non-full leaf node.

Table 11.3.1: 2-3-4 tree non-full-leaf insertion cases.

©zyBooks 06/21/23 23:08 169246
Taylor Larrechea
COLORADOCSPB2270Summer2023

Condition	Outcome
New key equals an existing key in node	No insertion takes place, and the node is not altered.
New key is < node's first key	Existing keys in node are shifted right, and the new key becomes node's first key.

Node has only 1 key or new key is < node's middle key	Node's middle key , if present, becomes last key, and new key becomes node's middle key.
None of the above	New key becomes node's last key.

PARTICIPATION ACTIVITY**11.3.5: Insertion of key into leaf node.**

©zyBooks 06/21/23 23:08 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

- 1) A non-full leaf node can have any key inserted.

- True
- False

- 2) When the key 30 is inserted into a leaf node with keys 20 and 40, 30 becomes which node value?

- A
- B
- C

- 3) When the key 50 is inserted into a leaf node with key 25, 50 becomes which node value?

- A
- B
- C

- 4) When inserting a new key into a node with 1 key, the new key can become the A, B, or C key in the node.

- True
- False

- 5) When the key 50 is inserted into a leaf node with keys 10, 20, and 30, 50 becomes which value?

- A
- B
- C

- none of the above

B-tree insert with preemptive split

Multiple insertion schemes exist for 2-3-4 trees. The **preemptive split** insertion scheme always splits any full node encountered during insertion traversal. The preemptive split insertion scheme ensures that any time a full node is split, the parent node has room to accommodate the middle value from the child.

PARTICIPATION ACTIVITY

11.3.6: B-tree insertion with preemptive split algorithm.



Animation content:

undefined

Animation captions:

1. Insertion of 60 starts at the root. A series of checks are executed on the node.
2. 60 is inserted and the root node is returned.
3. Insertion of 20 again begins at the root. The search ensures that 20 is not already in the node.
4. The full root node is split and the return value from the split is assigned to node.
5. The root node is not a leaf, so a recursive call is made to insert into the left child of the root.
6. After the series of checks, 20 is inserted and the left child of the root is returned.

PARTICIPATION ACTIVITY

11.3.7: Preemptive split insertion.



- 1) When arriving at a node during insertion, what is the first check that must take place?

- Check if the node is a leaf
- Check if the node already contains the key being inserted
- Check to see if the node is full

- 2) After any insertion operation completes, the root node will never have 3 keys.

- True

False

- 3) During insertion, a parent node can temporarily have 4 keys, if a child node is split.

 True False

- 4) If a node has 2 keys, 20 and 40, then only keys > 20 and < 40 could be inserted into this node.

©zyBooks 06/21/23 23:08 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

 True False

- 5) During insertion, how does a 2-3-4 expand in height?



- When a value is inserted into a leaf, the tree will always grow in height.
- When splitting a leaf node, the tree will always grow in height.
- When splitting the root node, the tree will always grow in height.
- Any insertion that does NOT involve splitting any nodes will cause the tree to grow in height.

CHALLENGE ACTIVITY

11.3.1: 2-3-4 tree insert algorithm.



489394.3384924.qx3zqy7

Start

©zyBooks 06/21/23 23:08 1692462

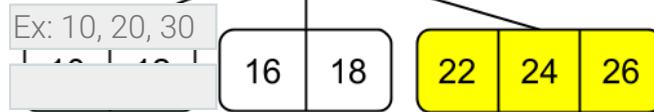
Taylor Larrechea

COLORADOCSPB2270Summer2023

The node (22, 24, 26) is split.

Enter each node's keys after the split, or "none" if the node doesn't exist.

Root:

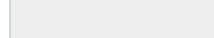


Root's left child:



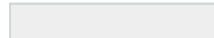
16 18

Root's middle1 child:

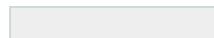


22 24 26

Root's middle2 child:



Root's right child:



©zyBooks 06/21/23 23:08 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Height of tree:

Ex: 5

1

2

3

4

Check

Next

11.4 2-3-4 tree rotations and fusion

Rotation concepts

Removing an item from a 2-3-4 tree may require rearranging keys to maintain tree properties. A **rotation** is a rearrangement of keys between 3 nodes that maintains all 2-3-4 tree properties in the process. The 2-3-4 tree removal algorithm uses rotations to transfer keys between sibling nodes. A **right rotation** on a node causes the node to lose one key and the node's right sibling to gain one key. A **left rotation** on a node causes the node to lose one key and the node's left sibling to gain one key.

PARTICIPATION
ACTIVITY

11.4.1: Left and right rotations.



Animation content:

©zyBooks 06/21/23 23:08 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

undefined

Animation captions:

1. A right rotation on the root's left child moves 23 into the root, and 27 into the root's middle1 child.

2. A left rotation on the root's right child moves 73 into the root, and 55 into the root's middle1 child.

PARTICIPATION ACTIVITY

11.4.2: 2-3-4 tree rotations.



1) A rotation on a node changes the set of keys in one of the node's children.

©zyBooks 06/21/23 23:08 169246

Taylor Larrechea

COLORADOCSPB2270Summer2023

 True False

2) A rotation on a node changes the set of keys in the node's parent.

 True False

3) A left rotation can only be performed on a node that has a left sibling.

 True False

4) A rotation operation may change the height of a 2-3-4 tree.

 True False

Utility functions for rotations

Several utility functions are used in the rotation operation.

- **BTreeGetLeftSibling** returns a pointer to the left sibling of a node or null if the node has no left sibling. BTreeGetLeftSibling returns null, left, middle1, or middle2 if the node is the left, middle1, middle2, or the right child of the parent, respectively. Since the parent node is required, a precondition of this function is that the node is not the root. ©zyBooks 06/21/23 23:08 169246
- **BTreeGetRightSibling** returns a pointer to the right sibling of a node or null if the node has no right sibling. ©zyBooks 06/21/23 23:08 169246
- **BTreeGetParentKeyLeftOfChild** takes a parent node and a child of the parent node as arguments, and returns the key in the parent that is immediately left of the child.
- **BTreeSetParentKeyLeftOfChild** takes a parent node, a child of the parent node, and a key as arguments, and sets the key in the parent that is immediately left of the child.

- **BTreeAddKeyAndChild** operates on a non-full node, adding one new key and one new child to the node. The new key must be greater than all keys in the node, and all keys in the new child subtree must be greater than the new key. Ex: If the node has 1 key, the newly added key becomes key B in the node, and the child becomes the middle2 child. If the node has 2 keys, the newly added key becomes key C in the node, and the child becomes the right child.
- **BTreeRemoveKey** removes a key from a node using a key index in the range [0,2]. This process may require moving keys and children to fill the location left by removing the key. The pseudocode for BTreeRemoveKey is below.

©zyBooks 06/21/23 23:08 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Figure 11.4.1: BTreeRemoveKey pseudocode.

```
BTreeRemoveKey(node, keyIndex)
{
    if (keyIndex == 0) {
        node->A = node->B
        node->B = node->C
        node->C = null
        node->left = node->middle1
        node->middle1 =
node->middle2
        node->middle2 = node->right
        node->right = null
    }
    else if (keyIndex == 1) {
        node->B = node->C
        node->C = null
        node->middle2 = node->right
        node->right = null
    }
    else if (keyIndex == 2) {
        node->C = null
        node->right = null
    }
}
```

PARTICIPATION ACTIVITY

11.4.3: Utility functions for rotations.



If unable to drag and drop, refresh the page.

©zyBooks 06/21/23 23:08 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

[BTreeSetParentKeyLeftOfChild](#)
[BTreeGetLeftSibling](#)
[BTreeAddKeyAndChild](#)
[BTreeGetRightSibling](#)
[BTreeGetParentKeyLeftOfChild](#)
[BTreeRemoveKey](#)

Removes a node's key by index.

Adds a new key and child into a node that has 1 or 2 keys.

Returns a pointer to a node's right-adjacent sibling.

©zyBooks 06/21/23 23:08 1692462

Taylor Larrechea

COLORADOCSBP2270Summer2023

Returns a pointer to a node's left-adjacent sibling.

Returns the key of the given parent that is immediately left of the given child.

Replaces the parent's key that is immediately left of the child with the specified key.

Reset

Rotation pseudocode

The rotation algorithm operates on a node, causing a net decrease of 1 key in that node. The key removed from the node moves up into the parent node, displacing a key in the parent that is moved to a sibling. No new nodes are allocated, nor existing nodes deallocated during rotation. The code simply copies key and child pointers.

PARTICIPATION
ACTIVITY

11.4.4: Left rotation pseudocode.



Animation content:

undefined

Animation captions:

©zyBooks 06/21/23 23:08 1692462

Taylor Larrechea

COLORADOCSBP2270Summer2023

1. A left rotation is performed on the root's middle1child. leftSibling is assigned with a pointer to node's left sibling, which is the root's left child.
2. keyForLeftSibling is assigned with 44, which is the key in parent's that is left of the node. Then, that key and the node's left child are added to the left sibling.
3. The node's leftmost key 66 is copied to the node's parent and then removed from the node.

PARTICIPATION ACTIVITY

11.4.5: Rotation Algorithm.



1) A rotation is a recursive operation.



- True
- False

2) A rotation will in some cases dynamically allocate a new node.

©zyBooks 06/21/23 23:08 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- True
- False

3) Any node that has an adjacent right sibling can be rotated right.



- True
- False

4) One child of the node being rotated will have a change of parent node.



- True
- False

Fusion

When rearranging values in a 2-3-4 tree during deletions, rotations are not an option for nodes that do not have a sibling with 2 or more keys. Fusion provides an additional option for increasing the number of keys in a node. A **fusion** is a combination of 3 keys: 2 from adjacent sibling nodes that have 1 key each, and a third from the parent of the siblings. Fusion is the inverse operation of a split. The key taken from the parent node must be the key that is between the 2 adjacent siblings. The parent node must have at least 2 keys, with the exception of the root.

Fusion of the root node is a special case that happens only when the root and the root's 2 children each have 1 key. In this case, the 3 keys from the 3 nodes are combined into a single node that becomes the new root node.

©zyBooks 06/21/23 23:08 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

11.4.6: Root fusion.



Animation content:

undefined

Animation captions:

1. Fusion of the root happens without allocating any new nodes. First, the A, B, and C keys are set to 41, 63, and 76, respectively.
2. The 4 child pointers of the root are copied from the child pointers of the 2 children.

PARTICIPATION
ACTIVITY

11.4.7: Root fusion.

©zyBooks 06/21/23 23:08 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



- 1) How many nodes are allocated in the root fusion pseudocode?

- 0
- 1
- 2
- 3



- 2) From where does the final B key in the root after fusion come?

- The A key in the root's left child.
- The A key in the root's right child.
- The original A key in the root.
- The original C key in the root.



- 3) How many keys will the root have after root fusion?

- 1
- 2
- 3
- 4



- 4) How many child pointers are changed in the root node during fusion?

- 0
- 2
- 3
- 4

©zyBooks 06/21/23 23:08 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



Non-root fusion

For the non-root case, fusion operates on 2 adjacent siblings that each have 1 key. The key in the parent node that is between the 2 adjacent siblings is combined with the 2 keys from the two siblings to make a single, fused node. The parent node must have at least 2 keys.

In the fusion algorithm below, the `BTreeGetKeyIndex` function returns an integer in the range [0,2] that indicates the index of the key within the node. The `BTreeSetChild` functions sets the left, middle1, middle2, or right child pointer based on an index value of 0, 1, 2, or 3, respectively.

PARTICIPATION ACTIVITY

11.4.8: Non-root fusion.

©zyBooks 06/21/23 23:08 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

**Animation content:**

undefined

Animation captions:

1. leftNode is the node with key 20 and rightNode is the node with key 54. The fuse operation starts by getting a pointer to the parent.
2. The parent node is root, but does not have 1 key, so `BTreeFuseRoot` is not called.
3. middleKey is assigned with 30, which is the parent's key between the left and right nodes' keys.
4. The fused node is allocated with keys 20, 30, and 54. The child pointers are assigned with the left and right node's children.
5. The parent's leftmost key and child are removed. Then the parent's left child pointer is assigned with fusedNode.

PARTICIPATION ACTIVITY

11.4.9: Non-root fusion.



- 1) If the parent of the node being fused is the root, then `BTreeFuseRoot` is called.

- True
- False

- 2) How many keys will the returned fused node have?

- 1
- 2
- 3
- Depends on the number of keys in the parent node

©zyBooks 06/21/23 23:08 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023





- 3) The leftmost key from the parent node is always moved down into the fused node.

- True
- False

- 4) When the parent node has a key removed, how many child pointers must be assigned with new values?

- Only 1
- At most 2
- 3 or 4
- 2, 3, or 4

©zyBooks 06/21/23 23:08 169246
Taylor Larrechea
COLORADOCSPB2270Summer2023

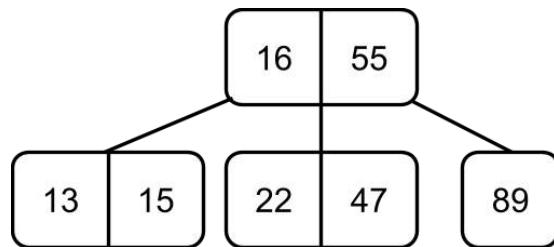
CHALLENGE ACTIVITY

11.4.1: 2-3-4 tree rotations and fusion.



489394.3384924.qx3zqy7

Start



A right rotation occurs on node (13, 15).

Enter each node's keys after the rotation, or **none** if the node doesn't exist.

Root:

Ex: 10, 20, 30, or none

©zyBooks 06/21/23 23:08 169246

Taylor Larrechea

COLORADOCSPB2270Summer2023

Root's left child:

Root's middle1 child:

Root's middle2 child:

Root's right child:

1

2

3

4

[Check](#)[Next](#)

©zyBooks 06/21/23 23:08 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

11.5 2-3-4 tree removal

Merge algorithm

A B-Tree **merge** operates on a node with 1 key and increases the node's keys to 2 or 3 using either a rotation or fusion. A node's 2 adjacent siblings are checked first during a merge, and if either has 2 or more keys, a key is transferred via a rotation. Such a rotation increases the number of keys in the merged node from 1 to 2. If all adjacent siblings of the node being merged have 1 key, then fusion is used to increase the number of keys in the node from 1 to 3. The merge operation can be performed on any node that has 1 key and a non-null parent node with at least 2 keys.

PARTICIPATION ACTIVITY

11.5.1: Merge algorithm.



Animation content:

undefined

Animation captions:

1. To merge the node with the key 25, a left rotation is performed on the right-adjacent sibling.
2. Since all siblings of the node with key 12 have 1 key, the merge operation is done with a fusion.

PARTICIPATION ACTIVITY

11.5.2: Merge algorithm.



If unable to drag and drop, refresh the page.

©zyBooks 06/21/23 23:08 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

[1, 2, or 3 keys](#)[2 or 3 keys](#)[Exactly 1 key](#)[Exactly 3 keys](#)

Number of keys a node must have to be merged.

Number of keys a node must have to transfer a key to an adjacent sibling during a merge.

Number of keys a node has after fusion.

After a node is merged, the parent of the node will be left with this number of keys.

Reset

Utility functions for removal

Several utility functions are used in a B-tree remove operation.

- **BTreeGetMinKey** returns the minimum key in a subtree.
- **BTreeGetChild** returns a pointer to a node's left, middle1, middle2, or right child, if the childIndex argument is 0, 1, 2, or 3, respectively.
- **BTreeNextNode** returns the child of a node that would be visited next in the traversal to search for the specified key.
- **BTreeKeySwap** swaps one key with another in a subtree. The replacement key must be known to be a key that can be used as a replacement without violating any of the 2-3-4 tree rules.

Figure 11.5.1: BTreeGetMinKey pseudocode.

```
BTreeGetMinKey(node) {
    cur = node
    while (cur->left != null)
    {
        cur = cur->left
    }
    return cur->A
}
```

©zyBooks 06/21/23 23:08 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Figure 11.5.2: BTreeGetChild pseudocode.

```
BTreeGetChild(node, childIndex)
{
    if (childIndex == 0)
        return node->left
    else if (childIndex == 1)
        return node->middle1
    else if (childIndex == 2)
        return node->middle2
    else if (childIndex == 3)
        return node->right
    else
        return null
}
```

©zyBooks 06/21/23 23:08 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Figure 11.5.3: BTreeNextNode pseudocode.

```
BTreeNextNode(node, key) {
    if (key < node->A)
        return node->left
    else if (node->B == null || key <
node->B)
        return node->middle1
    else if (node->C == null || key <
node->C)
        return node->middle2
    else
        return node->right
}
```

©zyBooks 06/21/23 23:08 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

```
BTreeKeySwap(node, existing, replacement) {
    if (node == null)
        return false

    keyIndex = BTreeGetKeyIndex(node, existing)
    if (keyIndex == -1) {
        next = BTreeNextNode(node, existing)
        return BTreeKeySwap(next, existing,
replacement)
    }

    if (keyIndex == 0)
        node->A = replacement
    else if (keyIndex == 1)
        node->B = replacement
    else
        node->C = replacement

    return true
}
```

©zyBooks 06/21/23 23:08 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

11.5.3: Utility functions for removal.



- 1) The BTreeGetMinKey function always returns the A key of a node.

- True
- False



- 2) The BTreeGetChild function returns null if the childIndex argument is greater than three or less than zero.

- True
- False



- 3) The BTreeNextNode function takes a key as an argument. The key argument will be compared to at most ____ keys in the node.

- 1
- 2
- 3
- 4



©zyBooks 06/21/23 23:08 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



4) What happens if the BTreeKeySwap function is called with an existing key parameter that does not reside in the subtree?

- The tree will not be changed and true will be returned.
- The tree will not be changed and false will be returned.
- The key in the tree that is closest to the existing key parameter will be replaced and true will be returned.
- The key in the tree that is closest to the existing key parameter will be replaced and false will be returned.

©zyBooks 06/21/23 23:08 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

5) The pseudocode for BTreeGetMinKey, BTreeGetChild, and BTreeNextNode have a precondition of the node parameter being non-null.



- True
- False

Remove algorithm

Given a key, a 2-3-4 tree **remove** operation removes the first-found matching key, restructuring the tree to preserve all 2-3-4 tree rules. Each successful removal results in a key being removed from a leaf node. Two cases are possible when removing a key, the first being that the key resides in a leaf node, and the second being that the key resides in an internal node.

A key can only be removed from a leaf node that has 2 or more keys. The **preemptive merge** removal scheme involves increasing the number of keys in all single-key, non-root nodes encountered during traversal. The merging always happens before any key removal is attempted. Preemptive merging ensures that any leaf node encountered during removal will have 2 or more keys, allowing a key to be removed from the leaf node without violating the 2-3-4 tree rules.

To remove a key from an internal node, the key to be removed is replaced with the minimum key in the right child subtree (known as the key's successor), or the maximum key in the leftmost child subtree. First, the key chosen for replacement is stored in a temporary variable, then the chosen key is removed recursively, and lastly the temporary key replaces the key to be removed.

PARTICIPATION ACTIVITY

11.5.4: BTreeRemove algorithm: leaf case.

**Animation content:**

undefined

Animation captions:

©zyBooks 06/21/23 23:08 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

1. Removal of 33 begins by traversing through the tree to find the key.
2. All single-key, non-root nodes encountered during traversal must be merged.
3. The key 33 is found in a leaf node and is removed by calling BTreeRemoveKey.

PARTICIPATION ACTIVITY

11.5.5: BTreeRemove algorithm: non-leaf case.

**Animation content:**

undefined

Animation captions:

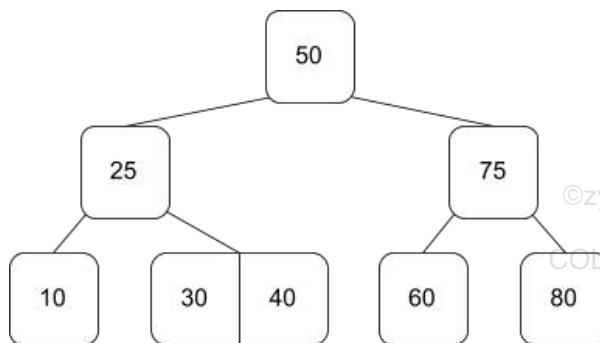
1. When deleting 60, the process is more complex due to the key being found in an internal node.
2. The key 62 is a suitable replacement for 60, but 62 must be recursively removed before the swap.
3. After the recursive removal completes, 60 is replaced with 62.

PARTICIPATION ACTIVITY

11.5.6: BTreeRemove algorithm.



Tree before removal:



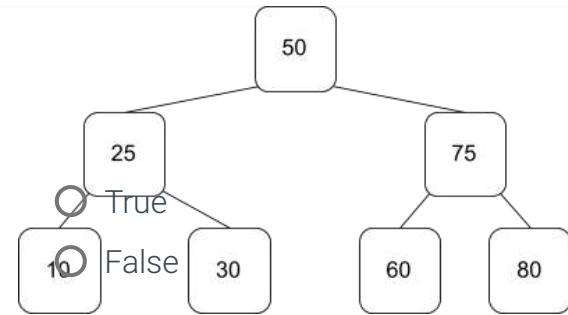
©zyBooks 06/21/23 23:08 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

- 1) The tree after removing 40 is:





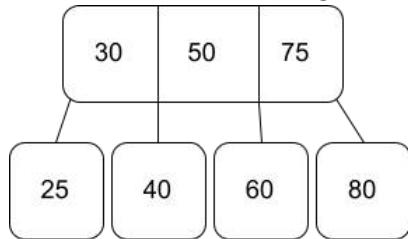
- 2) Calling BTreeRemove to remove any key in this tree would cause at least 1 node to be merged.

True
 False

- 3) Calling BTreeRemove to remove a key NOT in this tree would cause at least 1 node to be merged.

True
 False

- 4) The tree after removing 10 is:



True
 False

- 5) Calling BTreeRemove to remove key 50 would result in 75 being recursively removed and then used to replace 50.

True
 False



1) If a key in an internal node is to be removed, which key(s) in the tree may be used as replacements?

- Only the minimum key in right child subtree.
- Only the maximum key in left child subtree.
- Either the minimum key in the right child subtree or the maximum key in the left child subtree.
- Any adjacent key in the same node.

©zyBooks 06/21/23 23:08 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

2) During removal traversal, if the root node is encountered with 1 key, then the root node will be merged.

- True
- False



3) During removal traversal, any non-root node encountered with 1 key will be merged.

- True
- False

4) When removing a key in an internal node, a replacement key from elsewhere in the tree is chosen and stored in a temporary variable. What is true of the replacement key?

- The replacement key came from a leaf node.
- The replacement key is either the minimum or maximum key in the entire tree.

©zyBooks 06/21/23 23:08 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

- The replacement key will be swapped with the key to remove and then the replacement key will be recursively removed.
- No nodes will be merged during the recursive removal of the replacement key.

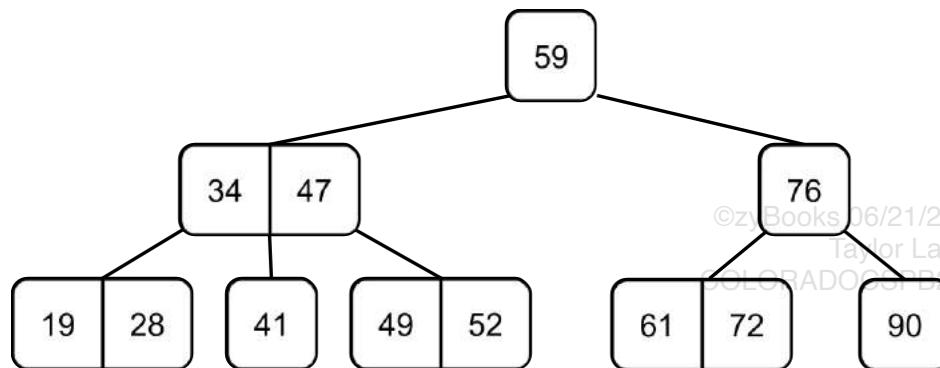
5) Removal pseudocode has the check: "if (keyIndex != -1)". What is implied about the node pointed to by cur when the condition evaluates to true?

- cur is null.
- cur has only 1 key.
- cur has no parent node.
- cur contains the key being removed.

CHALLENGE ACTIVITY

11.5.1: 2-3-4 tree removal.

489394.3384924.qx3zqy7

[Start](#)


A merge occurs on node (76).

Enter each node's keys after the merge, or **none** if the node doesn't exist:

©zyBooks 06/21/23 23:08 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Node	Keys
root	Ex: 10, 20, 30, or none
root → left	
root → middle1	
root → left → middle1	
root → middle1 → left	

©zyBooks 06/21/23 23:08 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

1

2

3

4

Check

Next

©zyBooks 06/21/23 23:08 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

12.1 Set abstract data type

Set abstract data type

A **set** is a collection of distinct elements. A set **add** operation adds an element to the set, provided an equal element doesn't already exist in the set. A set is an unordered collection. Ex: The set with integers 3, 7, and 9 is equivalent to the set with integers 9, 3 and 7.

PARTICIPATION
ACTIVITY

12.1.1: Set abstract data type.



Animation content:

undefined

Animation captions:

1. Adding 67, 91, and 14 produces a set with 3 elements.
2. Because 91 already exists in the set, adding 91 any number of additional times has no effect.
3. Set 2 is built by adding the same numbers in a different order.
4. Because order does not matter in a set, the 2 sets are equivalent.

PARTICIPATION
ACTIVITY

12.1.2: Set abstract data type.



- 1) Which of the following is not a valid set?



- { 78, 32, 46, 57, 82 }
- { 34, 8, 92 }
- { 78, 28, 91, 28, 15 }

- 2) How many elements are in a set that is built by adding the element 28 6 times, then the element 54 9 times?



- 1
- 2
- 15

- 3) Which 2 sets are equivalent?



- { 56, 19, 71 } and { 19, 65, 71, 56 }
- { 88, 54, 81 } and { 81, 88, 54 }
- { 39, 56, 14, 11 } and { 14, 56, 93, 11 }

©zyBooks 06/21/23 23:51 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Element keys and removal

Set elements may be primitive data values, such as numbers or strings, or objects with numerous data members. When storing objects, set implementations commonly distinguish elements based on an element's **key value**: A primitive data value that serves as a unique identifier for the element. Ex: An object for a student at a university may store information such as name, phone number, and ID number. No two students will have the same ID number, so the ID number can be used as the student object's key.

Sets are commonly implemented to use keys for all element types. When storing objects, the set retrieves an object's key via an external function or predetermined knowledge of which object property is the key value. When storing primitive data values, each primitive data value's key is itself.

Given a key, a set **remove** operation removes the element with the specified key from the set.

PARTICIPATION ACTIVITY

12.1.3: Element keys and removal.



Animation content:

undefined

Animation captions:

1. Different students at the same university may have the same name or phone number, but each student has a unique ID number.
2. A set for the course roster uses the student ID as the key value, since the exact same student cannot enroll twice in the same course.
3. The call to remove Student C provides only the student ID.

©zyBooks 06/21/23 23:51 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

12.1.4: Element keys and removal.



Refer to the example in the animation above.



- 1) If the student objects contained a field for GPA, then GPA could be used as the key value instead of student ID.

True
 False

- 2) `SetRemove(courseRosterSet, "Student D")` would remove Student D from the set.

©zyBooks 06/21/23 23:51 169246
Taylor Larrechea
COLORADOCSPB2270Summer2023

True
 False

- 3) SetRemove will not operate properly on an empty set.

True
 False

Searching and subsets

Given a key, a set **search** operation returns the set element with the specified key, or null if no such element exists. The search operation can be used to implement a subset test. A set X is a **subset** of set Y only if every element of X is also an element of Y.

PARTICIPATION ACTIVITY

12.1.5: SetIsSubset algorithm.



Animation content:

undefined

Animation captions:

- To test if set2 is a subset of set1, each element of set 2 is searched for in set1. Elements 19, 22, and 26 are found in set1.
- Element 34 is in set2 but not set1, so set2 is not a subset of set1.
- The first element in set3, 88, is not in set1, so set3 is not a subset of set1.
- All elements of set4 are in set1, so set4 is a subset of set1.
- No other set is a subset of another.
- But each set is always a subset of itself.

©zyBooks 06/21/23 23:51 169246
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

12.1.6: Searching and subsets.



1) Every set is a subset of itself.

True

False

2) For X to be a subset of Y, the number of elements in Y must be greater than or equal to the number of elements in X.

True

False

3) The loop in SetIsSubset always performs N iterations, where N is the number of elements in subsetCandidate.

True

False

CHALLENGE ACTIVITY

12.1.1: Set abstract data type.

489394.3384924.qx3zqy7

Start

Given an empty set numSet, what is numSet after the following operations?

SetAdd(numSet, 89)

SetAdd(numSet, 43)

SetAdd(numSet, 89)

SetAdd(numSet, 89)

SetAdd(numSet, 43)

SetAdd(numSet, 50)

{ Ex: 1, 2, 3 }

©zyBooks 06/21/23 23:51 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

1

2

3

Check

Next

12.2 Set operations

Union, intersection, and difference

The **union** of sets X and Y, denoted as $X \cup Y$, is a set that contains every element from X, every element from Y, and no additional elements. Ex: $\{ 54, 19, 75 \} \cup \{ 75, 12 \} = \{ 12, 19, 54, 75 \}$.

The **intersection** of sets X and Y, denoted as $X \cap Y$, is a set that contains every element that is in both X and Y, and no additional elements. Ex: $\{ 54, 19, 75 \} \cap \{ 75, 12 \} = \{ 75 \}$.

The **difference** of sets X and Y, denoted as $X \setminus Y$, is a set that contains every element that is in X but not in Y, and no additional elements. Ex: $\{ 54, 19, 75 \} \setminus \{ 75, 12 \} = \{ 54, 19 \}$.

The union and intersection operations are commutative, so $X \cup Y = Y \cup X$ and $X \cap Y = Y \cap X$. The difference operation is not commutative.

PARTICIPATION ACTIVITY

12.2.1: Set union, intersection, and difference.

**Animation content:**

undefined

Animation captions:

1. The union operation begins by adding all elements from set1.
2. Each element from set2 is added. Adding elements 82 and 93 has no effect, since 82 and 93 already exist in the result set.
3. The intersection operation iterates through each element in set1. Each element that is also in set2 is added to the result.
4. The difference of set1 and set2, denoted $set1 \setminus set2$, iterates through all elements in set1. Only elements 61 and 76 are added to the result, since these elements are not in set2.
5. Set difference is not commutative. $SetDifference(set2, set1)$ produces a result containing only 23 and 46, since those elements are in set2 but not in set1.

PARTICIPATION ACTIVITY

12.2.2: Union, intersection, and difference.

©zyBooks 06/21/23 23:51 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



- 1) How many elements are in the set $\{ 83, 5 \} \cup \{ 9, 77, 83 \}$?



- 2
 4

5

2) How many elements are in the set $\{ 83, 5 \} \cap \{ 9, 77, 83 \}$? □

 1 2 3

3) $\{ 83, 5 \} \setminus \{ 9, 77, 83 \} = ?$ □

 $\{ 83 \}$ $\{ 5 \}$ $\{ 83, 5 \}$

4) $\{ 9, 77, 83 \} \setminus \{ 83, 5 \} = ?$ □

 $\{ 9, 77 \}$ $\{ 9, 77, 83 \}$ $\{ 5 \}$

5) Which set operation is not commutative? □

 Union Intersection Difference

6) When X and Y do not have any elements in common, which is always true? □

 $X \cup Y = X \cap Y$ $X \cap Y = X \setminus Y$ $X \setminus Y = X$

7) Which is true for any set X? □

 $X \cup X = X \cap X$ $X \cup X = X \setminus X$ $X \setminus X = X \cap X$

©zyBooks 06/21/23 23:51 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Filter and map

A **filter** operation on set X produces a subset containing only elements from X that satisfy a particular condition. The condition for filtering is commonly represented by a **filter predicate**: A function that takes an element as an argument and returns a Boolean value indicating whether or not that element will be in the filtered subset.

A **map** operation on set X produces a new set by applying some function F to each element. Ex: If $X = \{18, 44, 38, 6\}$ and F is a function that divides a value by 2, then $\text{SetMap}(X, F) = \{9, 22, 19, 3\}$.

PARTICIPATION ACTIVITY

12.2.3: SetFilter and SetMap algorithms.

©zyBooks 06/21/23 23:51 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

**Animation content:**

undefined

Animation captions:

1. SetFilter is called with the EvenPredicate function passed as the second argument.
2. SetFilter calls EvenPredicate for each element. EvenPredicate returns true for each even element, and false for each odd element.
3. Every element for which the predicate returned true is added to the result, producing the set of even numbers from set1.
4. SetFilter(set1, Above90Predicate) produces the set with all elements from set1 that are greater than 90.
5. SetMap is called with the OnesDigit function passed as the first argument. Like SetFilter, SetMap calls the function for each element.
6. The returned value from each OnesDigit call is added to the result set, producing the set of distinct ones digit values.
7. SetMap(set1, StringifyElement) produces a set of strings from a set of numbers.

PARTICIPATION ACTIVITY

12.2.4: Using SetFilter with a set of strings.



Suppose $\text{stringSet} = \{\text{"zyBooks"}, \text{"Computer science"}, \text{"Data structures"}, \text{"set"}, \text{"filter"}, \text{"map"}\}$.

Filter predicates are defined below. Match each SetFilter call to the resulting set.

©zyBooks 06/21/23 23:51 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

```

StartsWithCapital(string) {
    if (string starts with capital letter) {
        return true
    }
    else {
        return false
    }
}

Has6OrFewerCharacters(string) {
    if (length of string <= 6) {
        return true
    }
    else {
        return false
    }
}

EndsInS(string) {
    if (string ends in "S" or "s") {
        return true
    }
    else {
        return false
    }
}

```

©zyBooks 06/21/23 23:51 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

If unable to drag and drop, refresh the page.

SetFilter(stringSet, StartsWithCapital)

SetFilter(stringSet, EndsInS)

SetFilter(stringSet, Has6OrFewerCharacters)

{ "zyBooks", "Data structures" }

{ "Computer science", "Data structures" }

{ "set", "filter", "map" }

©zyBooks 06/21/23 23:51 1692462

Reset

Taylor Larrechea

COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

12.2.5: Using SetMap with a set of numbers.



Suppose numbersSet = { 6.5, 4.2, 7.3, 9.0, 8.7 }. Map functions are defined below. Match each SetMap call to the resulting set.

```
MultiplyBy10(number) {  
    return number * 10.0  
}  
  
Floor(number) {  
    return floor(number)  
}  
  
Round(number) {  
    return round(number)  
}
```

©zyBooks 06/21/23 23:51 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

If unable to drag and drop, refresh the page.

SetMap(numbersSet, Round)

SetMap(numbersSet, MultiplyBy10)

SetMap(numbersSet, Floor)

{ 65.0, 42.0, 73.0, 90.0, 87.0 }

{ 7.0, 4.0, 9.0 }

{ 6.0, 4.0, 7.0, 9.0, 8.0 }

Reset

PARTICIPATION ACTIVITY

12.2.6: SetFilter and SetMap algorithm concepts.



- 1) A filter predicate must return true for elements that are to be added to the resulting set, and false for elements that are not to be added.

- True
- False



- 2) Calling SetFilter on set X always produces a set with the same number of elements as X.

- True
- False

©zyBooks 06/21/23 23:51 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023





3) Calling SetMap on set X always produces a set with the same number of elements as X.

- True
- False

4) Both SetFilter and SetMap will call the function passed as the second argument for every element in the set.

- True
- False

©zyBooks 06/21/23 23:51 169246
Taylor Larrechea
COLORADOCSPB2270Summer2023

CHALLENGE ACTIVITY

12.2.1: Set operations.



489394.3384924.qx3zqy7

Start

Given:

setA = { 39, 27, 22 }
setB = { 70, 27, 81 }

What is SetUnion(setA, setB)?

{ [Ex: 1, 2, 3] }

What is SetIntersection(setA, setB)?

{ [] }

1

2

3

4

©zyBooks 06/21/23 23:51 169246
Taylor Larrechea
COLORADOCSPB2270Summer2023

Check

Next

12.3 Static and dynamic set operations

A **dynamic set** is a set that can change after being constructed. A **static set** is a set that doesn't change after being constructed. A collection of elements is commonly provided during construction of a static set, each of which is added to the set. Ex: A static set constructed from the list of integers (19, 67, 77, 67, 59, 19) would be { 19, 67, 77, 59 }.

Static sets support most set operations by returning a new set representing the operation's result. The table below summarizes the common operations for static and dynamic sets.

©zyBooks 06/21/23 23:51 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Table 12.3.1: Static and dynamic set operations.

Operation	Dynamic set support?	Static set support?
Construction from a collection of values	Yes	Yes
Count number of elements	Yes	Yes
Search	Yes	Yes
Add element	Yes	No
Remove element	Yes	No
Union (returns new set)	Yes	Yes
Intersection (returns new set)	Yes	Yes
Difference (returns new set)	Yes	Yes
Filter (returns new set)	Yes	Yes
Map (returns new set)	Yes	Yes

PARTICIPATION ACTIVITY

12.3.1: Static and dynamic set operations.

©zyBooks 06/21/23 23:51 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

- 1) Static sets do not support union or intersection, since these operations require changing the set.

- True
- False



2) A static set constructed from the list of integers (20, 12, 87, 12) would be { 20, 12, 87, 12 }.

- True
- False

3) Suppose a dynamic set has N elements. Adding any element X and then removing element X will always result in the set still having N elements.

- True
- False

©zyBooks 06/21/23 23:51 169246

Taylor Larrechea

COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

12.3.2: Choosing static or dynamic sets for real-world datasets.



For each real-world dataset, select whether a program should use a static or dynamic set.

1) Periodic table of elements



- Static
- Dynamic

2) Collection of names of all countries on the planet



- Static
- Dynamic

3) List of contacts for a user



- Static
- Dynamic

©zyBooks 06/21/23 23:51 169246

Taylor Larrechea

COLORADOCSPB2270Summer2023

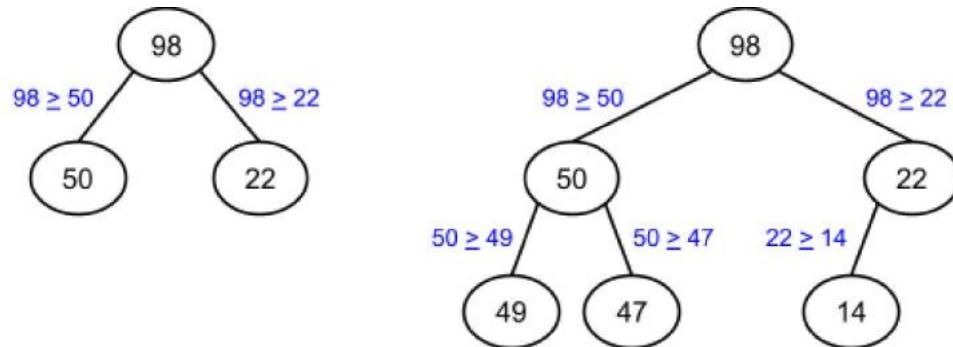
13.1 Heaps

Heap concept

Some applications require fast access to and removal of the maximum item in a changing set of items. For example, a computer may execute jobs one at a time; upon finishing a job, the computer executes the pending job having maximum priority. Ex: Four pending jobs have priorities 22, 14, 98, and 50; the computer should execute 98, then 50, then 22, and finally 14. New jobs may arrive at any time.

Maintaining jobs in fully-sorted order requires more operations than necessary, since only the maximum item is needed. A **max-heap** is a complete binary tree that maintains the simple property that a node's key is greater than or equal to the node's children's keys. (Actually, a max-heap may be any tree, but is commonly a binary tree). Because $x \geq y$ and $y \geq z$ implies $x \geq z$, the property results in a node's key being greater than or equal to all the node's descendants' keys. Therefore, a *max-heap's root always has the maximum key in the entire tree*.

Figure 13.1.1: Max-heap property: A node's key is greater than or equal to the node's children's keys.

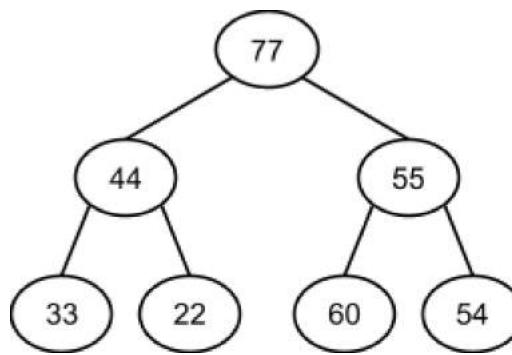


PARTICIPATION
ACTIVITY

13.1.1: Max-heap property.



Consider this binary tree:



©zyBooks 07/11/23 09:39 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



- 1) 33 violates the max-heap property due to being greater than 22.

- True
- False

- 2) 54 violates the max-heap property due to being greater than 44.

- True
- False

- 3) 60 violates the max-heap property due to being greater than 55.

- True
- False

- 4) A max-heap's root must have the maximum key.

- True
- False



Max-heap insert and remove operations

An **insert** into a max-heap starts by inserting the node in the tree's last level, and then swapping the node with its parent until no max-heap property violation occurs. Inserts fill a level (left-to-right) before adding another level, so the tree's height is always the minimum possible. The upward movement of a node in a max-heap is called **percolating**.

A **remove** from a max-heap is always a removal of the root, and is done by replacing the root with the last level's last node, and swapping that node with its greatest child until no max-heap property violation occurs. Because upon completion that node will occupy another node's location (which was swapped upwards), the tree height remains the minimum possible.



Animation captions:

1. This tree is a max-heap. A new node gets initially inserted in the last level...
2. ...and then percolate node up until the max-heap property isn't violated.
3. Removing a node (always the root): Replace with last node, then percolate node down.

PARTICIPATION
ACTIVITY

13.1.3: Max-heap inserts and deletes.

©zyBooks 07/11/23 09:39 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



- 1) Given N nodes, what is the height of a max-heap?

-
- N
- Depends on the keys



- 2) Given a max-heap with levels 0, 1, 2, and 3, with the last level not full, after inserting a new node, what is the maximum possible swaps needed?

- 1
- 2
- 3



- 3) Given a max-heap with N nodes, what is the worst-case complexity of an insert, assuming an insert is dominated by the swaps?

- O()
- O()



- 4) Given a max-heap with N nodes, what is the complexity for removing the root?

- O()
- O()

©zyBooks 07/11/23 09:39 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



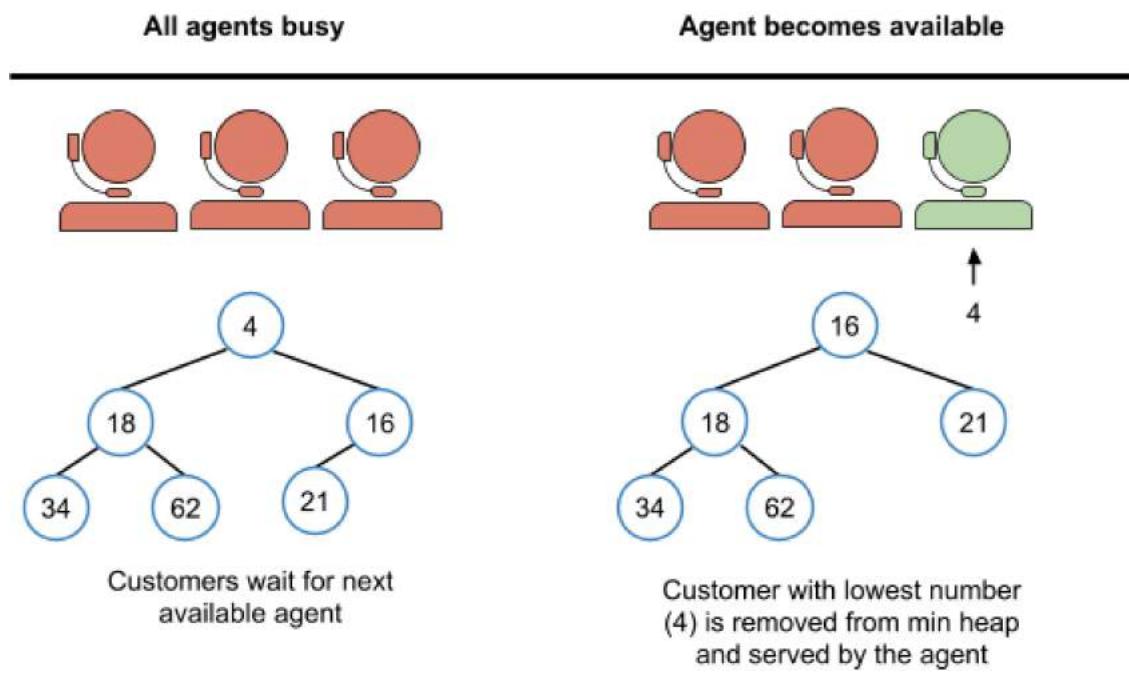
Min-heap

A **min-heap** is similar to a max-heap, but a node's key is less than or equal to its children's keys.

Example 13.1.1: Online tech support waiting lines commonly use min-heaps.

Many companies have online technical support that lets a customer chat with a support agent. If the number of customers seeking support is greater than the number of available agents, customers enter a virtual waiting line. Each customer has a priority that determines their place in line. The customer with the highest priority is served by the next available agent.

A min-heap is commonly used to manage prioritized queues of customers awaiting support. Customers that entered the line earlier and/or have a more urgent issue get assigned a lower number, which corresponds to a higher priority. When an agent becomes available, the customer with the lowest number is removed from the heap and served by the agent.



PARTICIPATION ACTIVITY

13.1.4: Min-heaps and customer support.



- 1) A customer with a higher priority has a lower numerical value in the min-heap.



- True
- False

- 2) If 2,000 customers are waiting for technical support, removing a customer from the min-heap requires about 2,000 operations.



©zyBooks 07/11/23 09:39 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

True False**CHALLENGE ACTIVITY**

13.1.1: Heaps.



489394.3384924.qx3zqy7

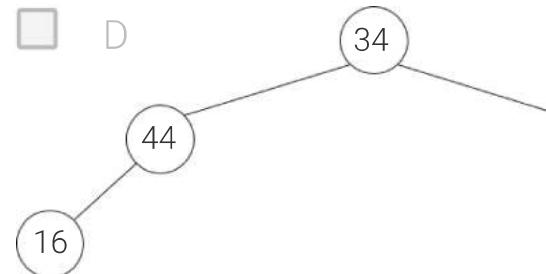
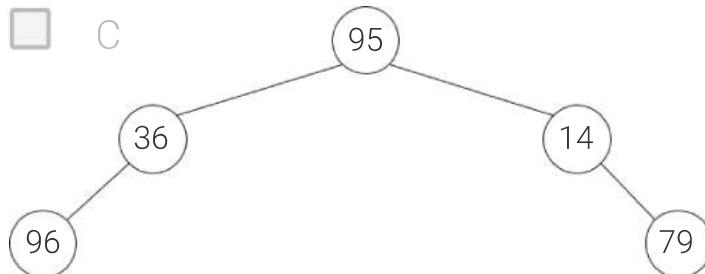
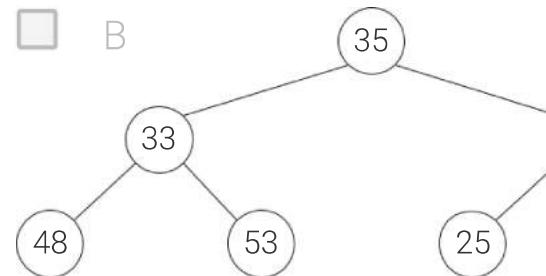
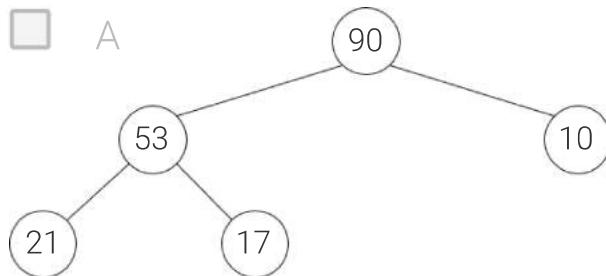
Start

©zyBooks 07/11/23 09:39 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Select all valid max-heaps.



1

2

3

4

Check**Next**

©zyBooks 07/11/23 09:39 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



13.2 Heaps using arrays

Heap storage

Heaps are typically stored using arrays. Given a tree representation of a heap, the heap's array form is produced by traversing the tree's levels from left to right and top to bottom. The root node is always the entry at index 0 in the array, the root's left child is the entry at index 1, the root's right child is the entry at index 2, and so on.

PARTICIPATION
ACTIVITY

13.2.1: Max-heap stored using an array.

©zyBooks 07/11/23 09:39 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



Animation captions:

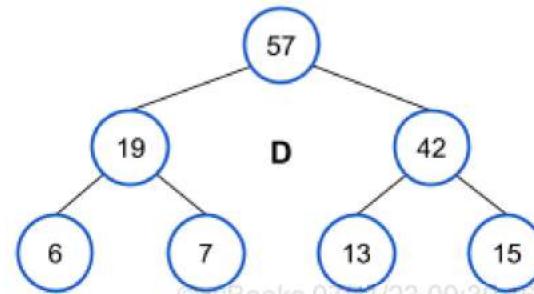
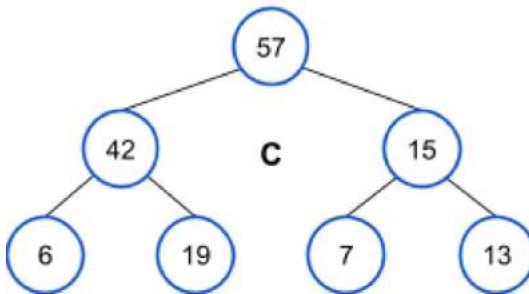
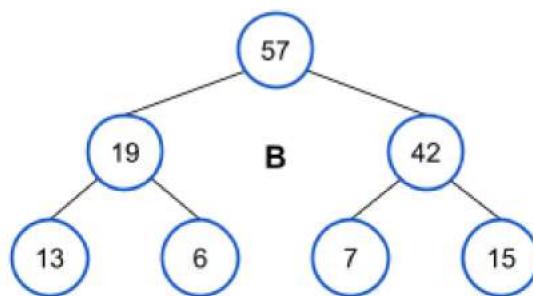
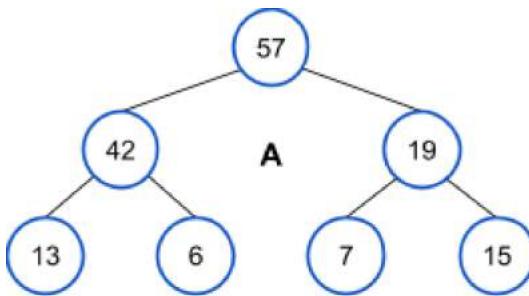
1. The max-heap's array form is produced by traversing levels left to right and top to bottom.
2. When 63 is inserted, the percolate-up operation happens within the array.

PARTICIPATION
ACTIVITY

13.2.2: Heap storage.



Match each max-heap to the corresponding storage array.



If unable to drag and drop, refresh the page.

©zyBooks 07/11/23 09:39 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

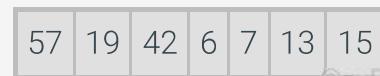
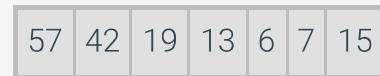
Heap A

Heap B

Heap D

Heap C

57	19	42	13	6	7	15
----	----	----	----	---	---	----



©zyBooks 07/11/23 09:39 1692462

Taylor Larrechea

COLORADO SPB2270Summer2023

Reset

Parent and child indices

Because heaps are not implemented with node structures and parent/child pointers, traversing from a node to parent or child nodes requires referring to nodes by index. The table below shows parent and child index formulas for a heap.

Table 13.2.1: Parent and child indices for a heap.

Node index	Parent index	Child indices
0	N/A	1, 2
1	0	3, 4
2	0	5, 6
3	1	7, 8
4	1	9, 10
5	2	11, 12
...
i		$2 * i + 1, 2 * i + 2$

PARTICIPATION ACTIVITY

13.2.3: Heap parent and child indices.





- 1) What is the parent index for a node at index 12?

- 3
- 4
- 5
- 6

©zyBooks 07/11/23 09:39 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



- 2) What are the child indices for a node at index 6?

- 7 and 8
- 12 and 13
- 13 and 14
- 12 and 24



- 3) The formula for computing parent node index should not be used on the root node.

- True
- False



- 4) The formula for computing child node indices does not work on the root node.

- True
- False



- 5) The formula for computing a child index evaluates to -1 if the parent is a leaf node.

- True
- False

Percolate algorithm

©zyBooks 07/11/23 09:39 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Following is the pseudocode for the array-based percolate-up and percolate-down functions. The functions operate on an array that represents a max-heap and refer to nodes by array index.

Figure 13.2.1: Max-heap percolate up algorithm.

```
MaxHeapPercolateUp(nodeIndex, heapArray) {
    while (nodeIndex > 0) {
        parentIndex = (nodeIndex - 1) / 2
        if (heapArray[nodeIndex] <= heapArray[parentIndex])
            return
        else {
            swap heapArray[nodeIndex] and
            heapArray[parentIndex]
            nodeIndex = parentIndex
        }
    }
}
```

©zyBooks 07/11/23 09:39 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Figure 13.2.2: Max-heap percolate down algorithm.

```
MaxHeapPercolateDown(nodeIndex, heapArray, arraySize) {
    childIndex = 2 * nodeIndex + 1
    value = heapArray[nodeIndex]

    while (childIndex < arraySize) {
        // Find the max among the node and all the node's
        children
        maxValue = value
        maxIndex = -1
        for (i = 0; i < 2 && i + childIndex < arraySize; i++)
        {
            if (heapArray[i + childIndex] > maxValue) {
                maxValue = heapArray[i + childIndex]
                maxIndex = i + childIndex
            }
        }

        if (maxValue == value) {
            return
        }
        else {
            swap heapArray[nodeIndex] and heapArray[maxIndex]
            nodeIndex = maxIndex
            childIndex = 2 * nodeIndex + 1
        }
    }
}
```

©zyBooks 07/11/23 09:39 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION
ACTIVITY

13.2.4: Percolate algorithm.

- 1) MaxHeapPercolateUp works for a node index of 0.

True
 False

- 2) MaxHeapPercolateDown has a precondition that nodeIndex is < arraySize.

True
 False

- 3) MaxHeapPercolateDown checks the node's left child first, and immediately swaps the nodes if the left child has a greater key.

True
 False

- 4) In MaxHeapPercolateUp, the while loop's condition nodeIndex > 0 guarantees that parentIndex is ≥ 0 .

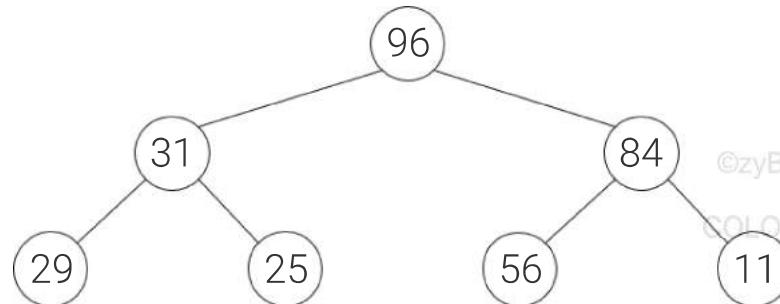
True
 False

CHALLENGE ACTIVITY

13.2.1: Heaps using arrays.

489394.3384924.qx3zqy7

Start



©zyBooks 07/11/23 09:39 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

What is the above max-heap's array form?

Ex: 86, 75, 30

(comma between values)

1	2	3	4	5
Check	Next			

©zyBooks 07/11/23 09:39 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

13.3 Heap sort

Heapify operation

Heapsort is a sorting algorithm that takes advantage of a max-heap's properties by repeatedly removing the max and building a sorted array in reverse order. An array of unsorted values must first be converted into a heap. The **heapify** operation is used to turn an array into a heap. Since leaf nodes already satisfy the max heap property, heapifying to build a max-heap is achieved by percolating down on every non-leaf node in reverse order.

**PARTICIPATION
ACTIVITY**

13.3.1: Heapify operation.



Animation captions:

1. If the original array is represented in tree form, the tree is not a valid max-heap.
2. Leaf nodes always satisfy the max heap property, since no child nodes exist that can contain larger keys. Heapification will start on node 92.
3. 92 is greater than 24 and 42, so percolating 92 down ends immediately.
4. Percolating 55 down results in a swap with 98.
5. Percolating 77 down involves a swap with 98. The resulting array is a valid max-heap.

The heapify operation starts on the internal node with the largest index and continues down to, and including, the root node at index 0. Given a binary tree with N nodes, the largest internal node index is

- 1.

©zyBooks 07/11/23 09:39 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Table 13.3.1: Max-heap largest internal node index.

Number of nodes in binary heap	Largest internal node index

1	-1 (no internal nodes)
2	0
3	0
4	1
5	1
6	2
7	2
...	...
N	-1

©zyBooks 07/11/23 09:39 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

13.3.2: Heapify operation.



- 1) For an array with 7 nodes, how many percolate-down operations are necessary to heapify the array?

Check
[Show answer](#)


- 2) For an array with 10 nodes, how many percolate-down operations are necessary to heapify the array?

Check
[Show answer](#)


©zyBooks 07/11/23 09:39 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

13.3.3: Heapify operation - critical thinking.



- 1) An array sorted in ascending order is already a valid max-heap.



True False

- 2) Which array could be heapified with the fewest number of operations, including all swaps used for percolating?

- (10, 20, 30, 40)
- (30, 20, 40, 10)
- (10, 10, 10, 10)

©zyBooks 07/11/23 09:39 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Heapsort overview

Heapsort begins by heapifying the array into a max-heap and initializing an end index value to the size of the array minus 1. Heapsort repeatedly removes the maximum value, stores that value at the end index, and decrements the end index. The removal loop repeats until the end index is 0.

PARTICIPATION ACTIVITY

13.3.4: Heapsort.



Animation captions:

1. The array is heapified first. Each internal node is percolated down, from highest node index to lowest.
2. The end index is initialized to 6, to refer to the last item. 94's "removal" starts by swapping with 68.
3. Removing from a heap means that the rightmost node on the lowest level disappears before the percolate down. End index is decremented after percolating.
4. 88 is swapped with 49, the last node disappears, and 49 is percolated down.
5. The process continues until end index is 0.
6. The array is sorted.

PARTICIPATION ACTIVITY

13.3.5: Heapsort.



Suppose the original array to be heapified is (11, 21, 12, 13, 19, 15).

©zyBooks 07/11/23 09:39 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

- 1) The percolate down operation must be performed on which nodes?

- 15, 19, and 13
- 12, 21, and 11
- All nodes in the heap



2) What are the first 2 elements swapped?

- 11 and 21
- 21 and 13
- 12 and 15

3) What are the last 2 elements swapped?

- 11 and 19
- 11 and 21
- 19 and 21

4) What is the heapified array?

- (11, 21, 12, 13, 19, 15)
- (21, 19, 15, 13, 12, 11)
- (21, 19, 15, 13, 11, 12)

©zyBooks 07/11/23 09:39 169246
Taylor Larrechea
COLORADOCSPB2270Summer2023

Heapsort algorithm

Heapsort uses 2 loops to sort an array. The first loop heapifies the array using MaxHeapPercolateDown. The second loop removes the maximum value, stores that value at the end index, and decrements the end index, until the end index is 0.

Figure 13.3.1: Heap sort.

```
Heapsort(numbers, numbersSize) {  
    // Heapify numbers array  
    for (i = numbersSize / 2 - 1; i >= 0; i--) {  
        MaxHeapPercolateDown(i, numbers,  
        numbersSize)  
    }  
  
    for (i = numbersSize - 1; i > 0; i--) {  
        Swap numbers[0] and numbers[i]  
        MaxHeapPercolateDown(0, numbers, i)  
    }  
}
```

©zyBooks 07/11/23 09:39 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

13.3.6: Heapsort algorithm.





- 1) How many times will MaxHeapPercolateDown be called by Heapsort when sorting an array with 10 elements?

- 5
- 10
- 14
- 20

©zyBooks 07/11/23 09:39 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 2) Calling Heapsort on an array with 1 element will cause an out of bounds array access.

- True
- False

- 3) Heapsort's worst-case runtime is $O(N \log N)$.

- True
- False

- 4) Heapsort uses recursion.

- True
- False



CHALLENGE ACTIVITY

13.3.1: Heap sort.



489394.3384924.qx3zqy7

Start

Given the array:

64	25	56	26	61	45	96
----	----	----	----	----	----	----

©zyBooks 07/11/23 09:39 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Heapify into a max-heap.

Ex: 86, 75, 30

1

2

3

[Check](#)[Next](#)

©zyBooks 07/11/23 09:39 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

13.4 Priority queue abstract data type (ADT)

Priority queue abstract data type

A **priority queue** is a queue where each item has a priority, and items with higher priority are closer to the front of the queue than items with lower priority. The priority queue **enqueue** operation inserts an item such that the item is closer to the front than all items of lower priority, and closer to the end than all items of equal or higher priority. The priority queue **dequeue** operation removes and returns the item at the front of the queue, which has the highest priority.

PARTICIPATION ACTIVITY

13.4.1: Priority queue enqueue and dequeue.



Animation content:

undefined

Animation captions:

1. Enqueueing a single item with priority 7 initializes the priority queue with 1 item.
2. If a lower numerical value indicates higher priority, enqueueing 11 adds the item to the end of the queue.
3. Since $5 < 7$, enqueueing 5 puts the item at the priority queue's front.
4. When enqueueing items of equal priority, the first-in-first-out rules apply. The 2nd item with priority 7 comes after the first.
5. Dequeue removes from the front of the queue, which is always the highest priority item.

©zyBooks 07/11/23 09:39 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023**PARTICIPATION ACTIVITY**

13.4.2: Priority queue enqueue and dequeue.



Assume that lower numbers have higher priority and that a priority queue currently holds items: 54, 71, 86 (front is 54).



- 1) Where would an item with priority 60 reside after being enqueued?
- Before 54
 - After 54
 - After 86

- 2) Where would an additional item with priority 54 reside after being enqueued?

©zyBooks 07/11/23 09:39 169246
Taylor Larrechea
COLORADOCSPB2270Summer2023

- Before the first 54
- After the first 54
- After 86

- 3) The dequeue operation would return which item?

- 54
- 71
- 86



Common priority queue operations

In addition to enqueue and dequeue, a priority queue usually supports peeking and length querying. A **peek** operation returns the highest priority item, without removing the item from the front of the queue.

Table 13.4.1: Common priority queue ADT operations.

Operation	Description	Example starting with priority queue: 42, 61, 98 (front is 42)
Enqueue(PQueue, x)	Inserts x after all equal or higher priority items	Enqueue(PQueue, 87). PQueue: 42, 61, 87, 98
Dequeue(PQueue)	Returns and removes the item at the front of PQueue	Dequeue(PQueue) returns 42. PQueue: 61, 98
Peek(PQueue)	Returns but does not remove the item at the front of PQueue	Peek(PQueue) returns 42. PQueue: 42, 61, 98

IsEmpty(PQueue)	Returns true if PQueue has no items	IsEmpty(PQueue) returns false.
GetLength(PQueue)	Returns the number of items in PQueue	GetLength(PQueue) returns 3.

©zyBooks 07/11/23 09:39 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

13.4.3: Common priority queue ADT operations.



Assume servicePQueue is a priority queue with contents: 11, 22, 33, 44, 55.

1) What does GetLength(servicePQueue) return?

- 5
- 11
- 55



2) What does Dequeue(servicePQueue) return?

- 5
- 11
- 55



3) After dequeuing an item, what will Peek(servicePQueue) return?

- 11
- 22
- 33



4) After calling Dequeue(servicePQueue) a total of 5 times, what will GetLength(servicePQueue) return?

- 1
- 0
- Undefined

©zyBooks 07/11/23 09:39 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Enqueueing items with priority

A priority queue can be implemented such that each item's priority can be determined from the item itself. Ex: A customer object may contain information about a customer, including the customer's name and a service priority number. In this case, the priority resides within the object.

A priority queue may also be implemented such that all priorities are specified during a call to **EnqueueWithPriority**: An enqueue operation that includes an argument for the enqueued item's priority.

©zyBooks 07/11/23 09:39 1692462

Taylor Larrechea



PARTICIPATION ACTIVITY

13.4.4: Priority queue EnqueueWithPriority operation

Animation content:

undefined

Animation captions:

1. Calls to EnqueueWithPriority() enqueue objects A, B, and C into the priority queue with the specified priorities.
2. In this implementation, the objects enqueued into the queue do not have data members representing priority.
3. Priorities specified during each EnqueueWithPriority() call are stored alongside the queue's objects.

PARTICIPATION ACTIVITY

13.4.5: EnqueueWithPriority operation.



- 1) A priority queue implementation that requires objects to have a data member storing priority would implement the _____ function.

- Enqueue
- EnqueueWithPriority



- 2) A priority queue implementation that does not require objects to have a data member storing priority would implement the _____ function.

- Enqueue
- EnqueueWithPriority

©zyBooks 07/11/23 09:39 1692462

Taylor Larrechea



COLORADO SPB2270 SUMMER 2023

Implementing priority queues with heaps

A priority queue is commonly implemented using a heap. A heap will keep the highest priority item in the root node and allow access in $O(1)$ time. Adding and removing items from the queue will operate in worst-case $O(\quad)$ time.

Table 13.4.2: Implementing priority queues with heaps.

Priority queue operation	Heap functionality used to implement operation	Worst-case runtime complexity
Enqueue	Insert	$O(\quad)$
Dequeue	Remove	$O(\quad)$
Peek	Return value in root node	$O(\quad)$
IsEmpty	Return true if no nodes in heap, false otherwise	$O(\quad)$
GetLength	Return number of nodes (expected to be stored in the heap's member data)	$O(\quad)$

PARTICIPATION ACTIVITY

13.4.6: Implementing priority queues with heaps.



- 1) The Dequeue and Peek operations both return the value in the root, and therefore have the same worst-case runtime complexity.



- True
- False

- 2) When implementing a priority queue with a heap, no operation will have a runtime complexity worse than $O(\quad)$.



- True
- False

- 3) If items in a priority queue with a lower numerical value have higher priority,



©zyBooks 07/11/23 09:39 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

then a max-heap should be used to implement the priority queue.

True

False

- 4) A priority queue is always implemented using a heap.

True

False



©zyBooks 07/11/23 09:39 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

CHALLENGE ACTIVITY

13.4.1: Priority queue abstract data type.



489394.3384924.qx3zqy7

Start

Assume that lower numbers have higher priority and that a priority queue numPQueue currently holds items: 11, 17, 35, 61 (front is 11).

Where does Enqueue(numPQueue, 10) add an item?



Where does Enqueue(numPQueue, 61) add an item?



Where does Enqueue(numPQueue, 12) add an item?



1

2

3

4

5

Check

Next

©zyBooks 07/11/23 09:39 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



13.5 Treaps

Treap basics

A BST built from inserts of N nodes having random-ordered keys stays well-balanced and thus has near-minimum height, meaning searches, inserts, and deletes are $O(\log N)$. Because insertion order may not be controllable, a data structure that somehow randomizes BST insertions is desirable. A **treap** uses a main key that maintains a binary search tree ordering property, and a secondary key generated randomly (often called "priority") during insertions that maintains a heap property. The combination usually keeps the tree balanced. The word "treap" is a mix of tree and heap. This section assumes the heap is a max-heap. Algorithms for basic treap operations include:

- A treap **search** is the same as a BST search using the main key, since the treap is a BST.
- A treap **insert** initially inserts a node as in a BST using the main key, then assigns a random priority to the node, and percolates the node up until the heap property is not violated. In a heap, a node is moved up via a swap with the node's parent. In a treap, a node is moved up via a *rotation at the parent*. Unlike a swap, a rotation maintains the BST property.
- A treap **delete** can be done by setting the node's priority such that the node should be a leaf ($-\infty$ for a max-heap), percolating the node down using rotations until the node is a leaf, and then removing the node.

PARTICIPATION ACTIVITY

13.5.1: Treap insert: First insert as a BST, then randomly assign a priority and use rotations to percolate node up to maintain heap.

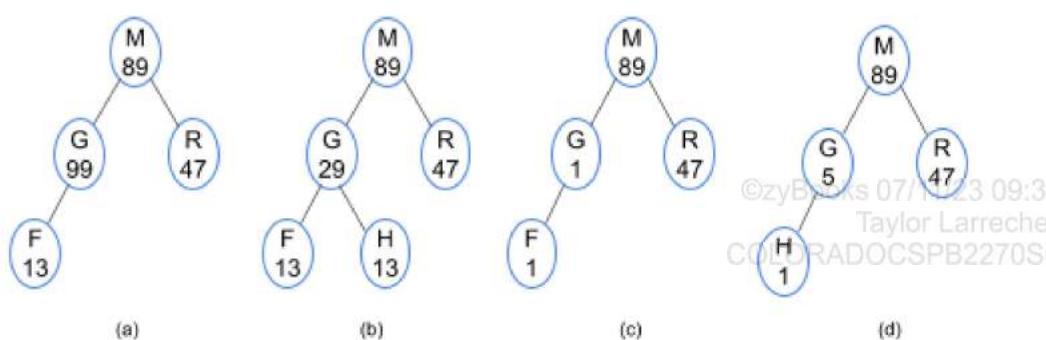


Animation captions:

1. The keys maintain a BST, the priorities a heap. Insert B as a BST...
2. Assign random priority (70). Rotate (which keep a BST) the node up until the priorities maintain a heap: 20 not > 70: Rotate. 47 not > 70: Rotate. 80 > 70: Done.

PARTICIPATION ACTIVITY

13.5.2: Recognizing treaps.



1) (a)



- Treap
 Not a treap

2) (b)



- Treap
- Not a treap

3) (c)



- Treap
- Not a treap

©zyBooks 07/11/23 09:39 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

4) (d)



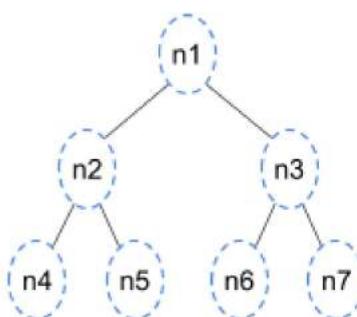
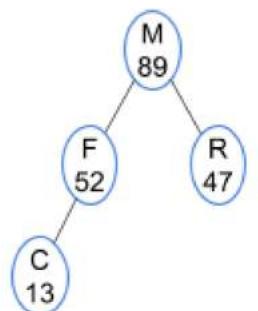
- Treap
- Not a treap

PARTICIPATION ACTIVITY

13.5.3: Treap insert.



When performing an insert, indicate each node's new location using the template tree's labels (n1...n7).



1) Where will a new node H first be inserted?

**Check****Show answer**

2) H is assigned a random priority of 20. To where does H percolate?

©zyBooks 07/11/23 09:39 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

**Check****Show answer**

- 3) Where will a new node P first be inserted?

Check**Show answer**

- 4) P is assigned a random priority of 65. To where does P percolate?

Check**Show answer**

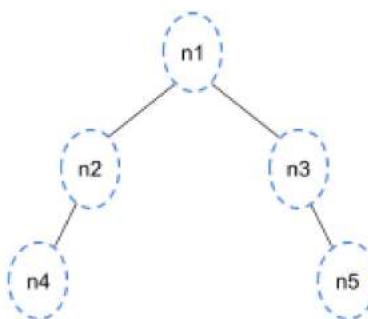
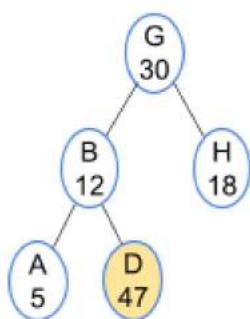
©zyBooks 07/11/23 09:39 169246
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

13.5.4: Treap insert.



Node D was just inserted, and assigned a random priority of 47. Rotations are needed to not violate the heap property. Match the node value to the corresponding location in the tree template on the right after the rotations are completed.



If unable to drag and drop, refresh the page.

D, 47**G, 30****H, 18****B, 12****A, 5**

n1

©zyBooks 07/11/23 09:39 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

n2

n3

n4

n5

Reset

Treap delete

A treap delete could be done by first doing a BST delete (copying the successor to the node-to-delete, then deleting the original successor), followed by percolating the node down until the heap property is not violated. However, a simpler approach just sets the node-to-delete's priority to $-\infty$ (for a max-heap), percolates the node down until a leaf, and removes the node. Percolating the node down uses rotations, not swaps, to maintain the BST property. Also, the node is rotated in the direction of the lower-priority child, so that the node rotated up has a higher priority than that child, to keep the heap property.

PARTICIPATION ACTIVITY

13.5.5: Treap delete: Set priority such that node must become a leaf, then percolate down using rotations.



Animation captions:

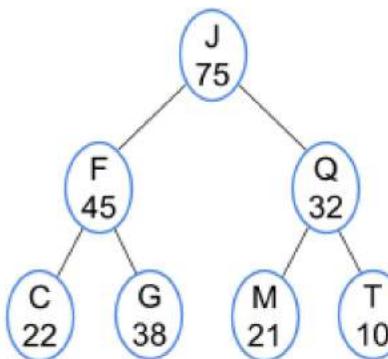
1. Node F is to be deleted. First set F's priority to $-\infty$.
2. Rotate (to keep a BST) until the node becomes a leaf node. $29 > 13$: Rotate right.
3. Rotate until node becomes a leaf node. Rotate left (the only option).
4. Remove leaf node.

PARTICIPATION ACTIVITY

13.5.6: Treap delete algorithm.



Each question starts from the original tree. Use this text notation for the tree: $(J (F (C, G), Q (M, T)))$. A - means the child does not exist.



©zyBooks 07/11/23 09:39 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 1) What is the tree after removing G?



- (J (F (C, -), Q (M, T)))
- (J (C (-, F), Q (M, T)))

2) What is the tree after removing Q?



- (J (F (C, G), M(-, T))
- (J (F (C, G), T(M, -))

**PARTICIPATION
ACTIVITY**

13.5.7: Treaps.

©zyBooks 07/11/23 09:39 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



1) A treap's nodes have random main keys.

- True
- False

2) A treap's nodes have random priorities.



- True
- False

3) Suppose a treap is built by inserting nodes with main keys in this order: A, B, C, D, E, F, G. The treap will have 7 levels, with each level having one node with a right child.



- True
- False

©zyBooks 07/11/23 09:39 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

14.1 Hash tables

Hash table overview

A **hash table** is a data structure that stores unordered items by mapping (or hashing) each item to a location in an array (or vector). Ex: Given an array with indices 0..9 to store integers from 0..500, the modulo (remainder) operator can be used to map 25 to index 5 ($25 \% 10 = 5$), and 149 to index 9 ($149 \% 10 = 9$). A hash table's main advantage is that searching (or inserting / removing) an item may require only $O(1)$, in contrast to $O(N)$ for searching a list or to $O(\log N)$ for binary search.

In a hash table, an item's **key** is the value used to map to an index. For all items that might possibly be stored in the hash table, every key is ideally unique, so that the hash table's algorithms can search for a specific item by that key.

Each hash table array element is called a **bucket**. A **hash function** computes a bucket index from the item's key.

PARTICIPATION ACTIVITY

14.1.1: Hash table data structure.



Animation captions:

1. A new hash table named playerNums with 10 buckets is created. A hash function maps an item's key to the bucket index.
2. A good hash function will distribute items into different buckets.
3. Hash tables provide fast search, using as few as one comparison.

PARTICIPATION ACTIVITY

14.1.2: Hash tables.



1) A 100 element hash table has 100 _____.

- items
- buckets



2) A hash function computes a bucket index from an item's _____.

- integer value
- key

©zyBooks 07/15/23 19:14 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023





- 3) For a well-designed hash table,
searching requires ____ on average.

- O(1)
- O(N)
- O(log N)

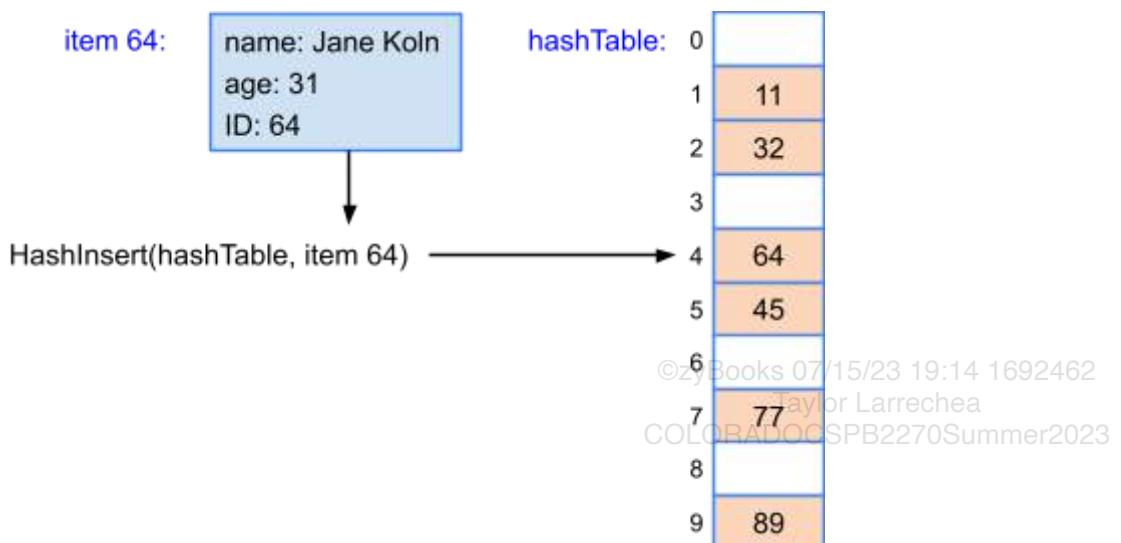
- 4) A company will store all employees in a hash table. Each employee item consists of a name, department, and employee ID number. Which is the most appropriate key?

- Name
- Department
- Employee ID number

©zyBooks 07/15/23 19:14 169246
Taylor Larrechea
COLORADOCSPB2270Summer2023

Item representation

Normally, each item being stored is an object with several fields, such as a person object having a name, age, and ID number, with the ID number used as the key. For simplicity, this section represents an item just by the item's key. Ex: Item 64 represents a person object with a key of 64, which is the person's ID. HashInsert(hashTable, item 64) inserts item 64 in bucket 4, representing item 64 in the hash table just by the key 64.



Hash table operations

A common hash function uses the **modulo operator %**, which computes the integer remainder when dividing two numbers. Ex: For a 20 element hash table, a hash function of key % 20 will map keys to bucket indices 0 to 19.

A hash table's operations of insert, remove, and search each use the hash function to determine an item's bucket. Ex: Inserting 113 first determines the bucket to be 113 % 20 = 3

PARTICIPATION
ACTIVITY

14.1.3: Hash tables.

3/15/23 19:14 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



- 1) A modulo hash function for a 50 entry hash table is: key % _____

Check

Show answer



- 2) key % 1000 maps to indices 0 to _____.

Check

Show answer



- 3) A modulo hash function is used to map to indices 0 to 9. The hash function should be: key % _____

Check

Show answer



- 4) Given a hash table with 100 buckets and modulo hash function, in which bucket will HashInsert(table, item 334) insert item 334?

Check

Show answer



©zyBooks 07/15/23 19:14 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



- 5) Given a hash table with 50 buckets and modulo hash function, in which bucket will HashSearch(table, 201) search for the item?

Check**Show answer**

©zyBooks 07/15/23 19:14 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY**14.1.4: Hash table search efficiency.**

Consider a modulo hash function (key % 10) and the following hash table.

numsTable:	0	
	11	
2	22	
3		
4		
5	45	
6		
7	47	
8		
9	39	

- 1) How many buckets will be checked for HashSearch(numsTable, 45)?

- 1
- 6
- 10

- 2) If item keys range from 0 to 49, how many keys may map to the same bucket?

- 1
- 5
- 50

©zyBooks 07/15/23 19:14 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



3) If a linear search were applied to the array, how many array elements would be checked to find item 45?

- 1
- 6
- 10

©zyBooks 07/15/23 19:14 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Empty cells

The approach for a hash table algorithm determining whether a cell is empty depends on the implementation. For example, if items are simply non-negative integers, empty can be represented as -1. More commonly, items are each an object with multiple fields (name, age, etc.), in which case each hash table array element may be a pointer. Using pointers, empty can be represented as null.

Collisions

A **collision** occurs when an item being inserted into a hash table maps to the same bucket as an existing item in the hash table. Ex: For a hash function of key % 10, 55 would be inserted in bucket 55 % 10 = 5; later inserting 75 would yield a collision because 75 % 10 is also 5. Various techniques are used to handle collisions during insertions, such as chaining or open addressing. **Chaining** is a collision resolution technique where each bucket has a list of items (so bucket 5's list would become 55, 75). **Open addressing** is a collision resolution technique where collisions are resolved by looking for an empty bucket elsewhere in the table (so 75 might be stored in bucket 6). Such techniques are discussed later in this material.

PARTICIPATION ACTIVITY

14.1.5: Hash table collisions.



1) A hash table's items will be positive integers, and -1 will represent empty. A 5-bucket hash table is: -1, -1, 72, 93, -1. How many items are in the table?

- 0
- 2
- 5

©zyBooks 07/15/23 19:14 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



2) A hash table has buckets 0 to 9 and uses a hash function of key % 10. If the table is initially empty and the following inserts are applied in the order shown, the insert of which item results in a collision?

HashInsert(hashTable, item 55)

HashInsert(hashTable, item 90)

HashInsert(hashTable, item 95)

©zyBooks 07/15/23 19:14 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

- Item 55
- Item 90
- Item 95

**CHALLENGE
ACTIVITY**

14.1.1: Hash tables with modulo hash function.



489394.3384924.qx3zqy7

Start

A hash table with non-negative integer keys has a modulo hash function of key % 25.

Hash function index range: 0 to Ex: 5

Item 193 will go in bucket Ex: 26

1

2

3

Check

Next

©zyBooks 07/15/23 19:14 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

14.2 Chaining

Chaining handles hash table collisions by using a list for each bucket, where each list may store multiple items that map to the same bucket. The insert operation first uses the item's key to determine the bucket, and then inserts the item in that bucket's list. Searching also first determines the bucket, and then searches the bucket's list. Likewise for removes.

PARTICIPATION ACTIVITY

14.2.1: Hash table with chaining.



©zyBooks 07/15/23 19:14 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Animation content:

undefined

Animation captions:

1. A hash table with chaining uses a list for each bucket. The insert operation first uses the item's key to determine the mapped bucket, and then inserts the item in that bucket's list.
2. A bucket may store multiple items with different keys that map to the same bucket. If collisions occur, items are inserted in the bucket's list.
3. Search first uses the item's key to determine the mapped bucket, and then searches the items in that bucket's list.

Figure 14.2.1: Hash table with chaining: Each bucket contains a list of items.

©zyBooks 07/15/23 19:14 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

```

HashInsert(hashTable, item) {
    if (HashSearch(hashTable, item->key) == null) {
        bucketList = hashTable[Hash(item->key)]
        node = Allocate new linked list node
        node->next = null
        node->data = item
        ListAppend(bucketList, node)
    }
}

HashRemove(hashTable, item) {
    bucketList = hashTable[Hash(item->key)]
    itemNode = ListSearch(bucketList, item->key)
    if (itemNode is not null) {
        ListRemove(bucketList, itemNode)
    }
}

HashSearch(hashTable, key) {
    bucketList = hashTable[Hash(key)]
    itemNode = ListSearch(bucketList, key)
    if (itemNode is not null)
        return itemNode->data
    else
        return null
}

```

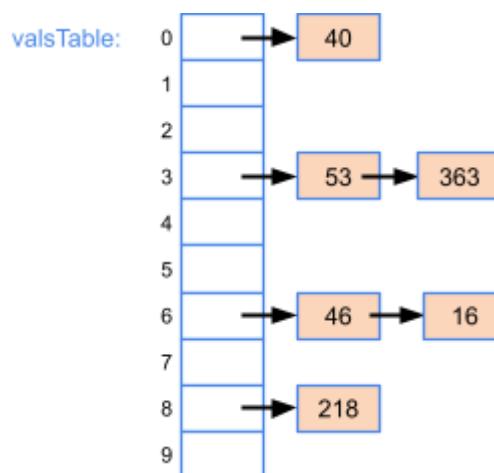
©zyBooks 07/15/23 19:14 1692462

Taylor Larrechea

COLORADOCSBP2270Summer2023

PARTICIPATION ACTIVITY**14.2.2: Hash table with chaining: Inserting items.**

Given hash function of key % 10, type the specified bucket's list after the indicated operation(s). Assume items are inserted at the end of a bucket's list. Type the bucket list as: 5, 7, 9 (or type: Empty).



©zyBooks 07/15/23 19:14 1692462

Taylor Larrechea

COLORADOCSBP2270Summer2023

- 1) HashInsert(valsTable, item 20)

Bucket 0's list: _____



/ /**Check****Show answer**

- 2) HashInsert(valsTable, item 23)
HashInsert(valsTable, item 99)
Bucket 3's list: _____

©zyBooks 07/15/23 19:14 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

/ /**Check****Show answer**

- 3) HashRemove(valsTable, 46)
Bucket 6's list: _____


/ /**Check****Show answer**

- 4) HashRemove(valsTable, 218)
Bucket 8's list: _____


/ /**Check****Show answer****PARTICIPATION ACTIVITY**

14.2.3: Hash table with chaining: Search.

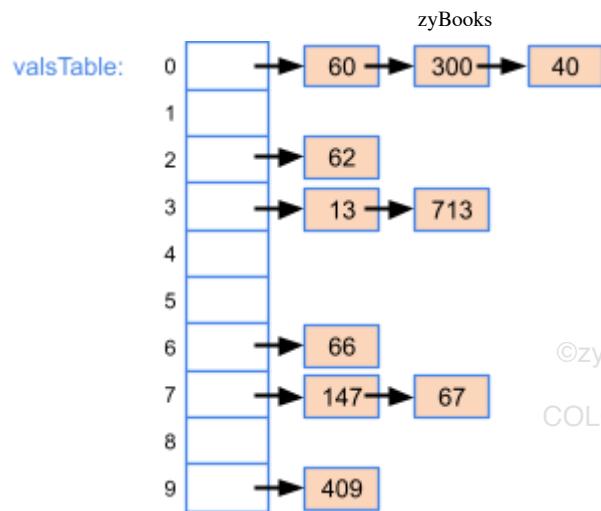


Consider the following hash table, and a hash function of key % 10.

©zyBooks 07/15/23 19:14 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



©zyBooks 07/15/23 19:14 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 1) How many list elements are compared for HashSearch(valsTable, 62)?

Check**Show answer**

- 2) How many list elements are compared for HashSearch(valsTable, 40)?

Check**Show answer**

- 3) What does HashSearch(valsTable, 186) return?

Check**Show answer**

- 4) How many list elements are compared for HashSearch(valsTable, 837)?

Check**Show answer**

©zyBooks 07/15/23 19:14 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

14.2.1: Chaining.



489394.3384924.qx3zqy7

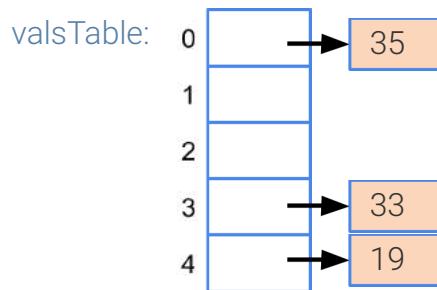
Start

Hash table valsTable is shown below. The hash function is key % 5. Assume items are inserted at the end of a bucket's list.

©zyBooks 07/15/23 19:14 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



After the operations

HashInsert(valsTable, item 93)

HashInsert(valsTable, item 16)

Select the bucket containing the following items:

- 33
- 93
- 16

**Check****Next**

14.3 Linear probing

©zyBooks 07/15/23 19:14 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Linear probing overview

A hash table with **linear probing** handles a collision by starting at the key's mapped bucket, and then linearly searches subsequent buckets until an empty bucket is found.

**PARTICIPATION
ACTIVITY**

14.3.1: Hash table with linear probing.



Animation captions:

1. During an insert, if a bucket is not empty, a collision occurs. Using linear probing, inserts will linearly probe buckets until an empty bucket is found.
2. The item is inserted in the next empty bucket.
3. Search starts at the hashed location and will compare each bucket until a match is found.
4. If an empty bucket is found, search returns null, indicating a matching item was not found.

©zyBooks 07/15/23 19:14 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

**PARTICIPATION
ACTIVITY**

14.3.2: Hash table with linear probing: Insert.



Given hash function of key % 5, determine the insert location for each item.

1) HashInsert(numsTable, item 13)

numsTable:	0	<input type="text"/>
	1	71
	2	22
	3	<input type="text"/>
	4	<input type="text"/>

bucket =

//
Check**Show answer**

2) HashInsert(numsTable, item 41)

numsTable:	0	<input type="text"/>
	1	21
	2	<input type="text"/>
	3	<input type="text"/>
	4	<input type="text"/>

bucket =

//
Check**Show answer**

3) HashInsert(numsTable, item 90)

numsTable:	0	50	<input type="text"/>
	1	31	<input type="text"/>
	2	<input type="text"/>	<input type="text"/>
	3	<input type="text"/>	<input type="text"/>
	4	4	<input type="text"/>

©zyBooks 07/15/23 19:14 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



bucket = //

Check**Show answer**

4) HashInsert(numsTable, item 74)

numsTable:	0	20
	1	
	2	32
	3	
	4	94

bucket = //

Check**Show answer**

©zyBooks 07/15/23 19:14 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Empty bucket types

Actually, linear probing distinguishes two types of empty buckets. An **empty-since-start** bucket has been empty since the hash table was created. An **empty-after-removal** bucket had an item removed that caused the bucket to now be empty. The distinction will be important during searches, since searching only stops for empty-since-start, not for empty-after-removal.

PARTICIPATION ACTIVITY

14.3.3: Hash with linear probing: Bucket status.



Given hash function of key % 10, determine the bucket status after the following operations have been executed.

HashInsert(valsTable, item 64)

HashInsert(valsTable, item 20)

HashInsert(valsTable, item 51)

HashRemove(valsTable, 51)

©zyBooks 07/15/23 19:14 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

1) Bucket 2

- empty-since-start
- empty-after-removal



2) Bucket 1



- empty-since-start
 - empty-after-removal
- 3) Bucket 4
- occupied
 - empty-after-removal

©zyBooks 07/15/23 19:14 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



Inserts using linear probing

Using linear probing, a hash table *insert* algorithm uses the item's key to determine the initial bucket, linearly probes (or checks) each bucket, and inserts the item in the next empty bucket (the empty kind doesn't matter). If the probing reaches the last bucket, the probing continues at bucket 0. The insert algorithm returns true if the item was inserted, and returns false if all buckets are occupied.

PARTICIPATION ACTIVITY

14.3.4: Insert with linear probing.



Animation content:

undefined

Animation captions:

1. Insert algorithm uses the item's key to determine the initial bucket.
2. Insert linearly probes (or checks) each bucket until an empty bucket is found.
3. Item is inserted into the next empty bucket.
4. If probing reaches the last bucket without finding an empty bucket, the probing continues at bucket 0.
5. Insert linearly probes each bucket until an empty bucket is found.

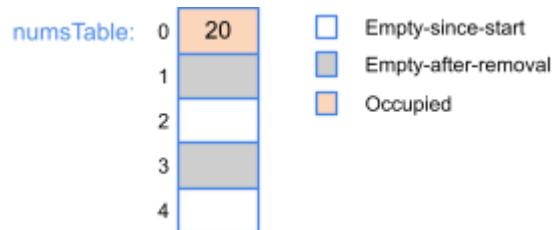
PARTICIPATION ACTIVITY

14.3.5: Hash table with linear probing: Insert with empty-after-removal buckets.

©zyBooks 07/15/23 19:14 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



For the given hash table and hash function of key % 5, what are the contents for each bucket after the following operations?



HashInsert(numsTable, item 43)
 HashInsert(numsTable, item 300)
 HashInsert(numsTable, item 71)

©zyBooks 07/15/23 19:14 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

If unable to drag and drop, refresh the page.

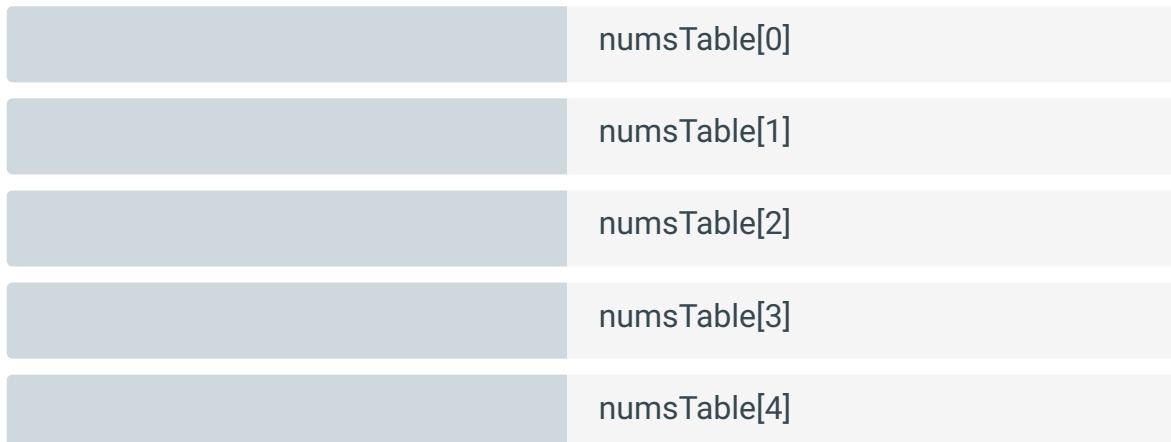
Item 71

Item 300

Item 43

Empty-since-start

Item 20



Reset

Removals using linear probing

Using linear probing, a hash table *remove* algorithm uses the sought item's key to determine the initial bucket. The algorithm probes each bucket until either a matching item is found, an empty-since-start bucket is found, or all buckets have been probed. If the item is found, the item is removed, and the bucket is marked empty-after-removal.

PARTICIPATION ACTIVITY

14.3.6: Remove with linear probing.

©zyBooks 07/15/23 19:14 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

Animation content:

undefined

Animation captions:

1. The remove algorithm uses the sought item's key to determine the initial bucket, probing buckets to find a matching item.
2. If the matching item is found, the item is removed, and the bucket is marked empty-after-removal.
3. Remove algorithm probes each bucket until either the matching item or an empty-since-start bucket is found.
4. If the matching item is found, the bucket is marked empty-after-removal.

©zyBooks 07/15/23 19:14 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Note that if the algorithm encounters an empty-after-removal bucket, the algorithm keeps probing, because the sought item may have been placed in a subsequent bucket before this bucket's item was removed. Ex: Removing item 42 above would start at bucket 2. Because bucket 2 is empty-after-removal, the algorithm would proceed to bucket 3, where item 42 would be found and removed.

PARTICIPATION ACTIVITY**14.3.7: Hash table with linear probing: Remove.**

Consider the following hash table and a hash function of key % 10.

idsTable:

0	20	Empty-since-start
1	68	Empty-after-removal
2	22	Occupied
3		
4	34	
5		
6	115	
7	65	
8	48	
9	199	

- 1) HashRemove(idsTable, 65) probes _____ buckets.



Check**Show answer**

- 2) HashRemove(idsTable, 10) probes _____ buckets.

©zyBooks 07/15/23 19:14 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Check**Show answer**



- 3) HashRemove(idsTable, 68) probes
_____ buckets.

[Show answer](#)

©zyBooks 07/15/23 19:14 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Searching using linear probing

In linear probing, a hash table search algorithm uses the sought item's key to determine the initial bucket. The algorithm probes each bucket until either the matching item is found (returning the item), an empty-since-start bucket is found (returning null), or all buckets are probed without a match (returning null). If an empty-after-removal bucket is found, the search algorithm continues to probe the next bucket.

PARTICIPATION ACTIVITY

14.3.8: Search with linear probing.



Animation content:

undefined

Animation captions:

1. The search algorithm uses the sought item's key to determine the initial bucket, and then linearly probes each bucket until a matching item is found.
2. If search reaches the last bucket without finding a matching item or empty-since-start bucket, the search continues at bucket 0.
3. If an empty-after-removal bucket is encountered, the algorithm continues to probe the next bucket.
4. If an empty-since-start bucket is encountered, the search algorithm returns null.

PARTICIPATION ACTIVITY

14.3.9: Hash table with linear probing: Search.



©zyBooks 07/15/23 19:14 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Consider the following hash table and a hash function of key % 10.

valsTable:	0	60
	1	
	2	110
	3	
	4	364
	5	75
	6	66
	7	
	8	
	9	49

- Empty-since-start
- Empty-after-removal
- Occupied

©zyBooks 07/15/23 19:14 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

- 1) HashSearch(valsTable, 75) probes
 ____ buckets.

Check**Show answer**

- 2) HashSearch(valsTable, 110) probes
 ____ buckets.

Check**Show answer**

- 3) What does HashSearch(valsTable,
 112) return?

Check**Show answer**

- 4) HashSearch(valsTable, 207) probes
 ____ buckets.

Check**Show answer**

©zyBooks 07/15/23 19:14 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

CHALLENGE ACTIVITY

14.3.1: Linear probing.

489394.3384924.qx3zqy7

Start

valsTable:	0	
	1	41
	2	
	3	
	4	
	5	
	6	76
	7	87
	8	
	9	

- Empty-since-start
- Empty-after-removal
- Occupied

©zyBooks 07/15/23 19:14 1692462

COLORADOCSPB2270Summer2023

HashInsert(valsTable, item 31) inserts item 31 into bucket Ex: 10HashInsert(valsTable, item 84) inserts item 84 into bucket HashInsert(valsTable, item 66) inserts item 66 into bucket

1

2

3

4

Check**Next**

14.4 Quadratic probing

Overview and insertion

A hash table with **quadratic probing** handles a collision by starting at the key's mapped bucket, and then quadratically searches subsequent buckets until an empty bucket is found. If an item's mapped bucket is H, the formula $H + (c_1i + c_2i^2)$ is used to determine the item's index in the hash table. c_1 and c_2 are programmer-defined constants for quadratic probing. Inserting a key uses the formula, starting with $i = 0$, to repeatedly search the hash table until an empty bucket is found. Each time an empty bucket is not found, i is incremented by 1. Iterating through sequential i values to obtain the desired table index is called the **probing sequence**.

©zyBooks 07/15/23 19:14 1692462

Taylor Larrechea

PARTICIPATION ACTIVITY14.4.1: Hash table insertion using quadratic probing: $c_1=1$ and $c_2=1$. Summer2023

Animation content:

undefined

Animation captions:

1. When inserting 55, no collision occurs with the first computed index of 5. Inserting 66 also does not cause a collision.
2. Inserting 25 causes a collision with the first computed index of 5.
3. i is incremented to 1 and a new index of 7 is computed. Bucket 7 is empty and 25 is inserted.

©zyBooks 07/15/23 19:14 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Figure 14.4.1: HashInsert with quadratic probing.

```
HashInsert(hashTable, item) {  
    i = 0  
    bucketsProbed = 0  
  
    // Hash function determines initial bucket  
    bucket = Hash(item->key) % N  
    while (bucketsProbed < N) {  
        // Insert item in next empty bucket  
        if (hashTable[bucket] is Empty) {  
            hashTable[bucket] = item  
            return true  
        }  
  
        // Increment i and recompute bucket index  
        // c1 and c2 are programmer-defined constants for quadratic  
        probing  
        i = i + 1  
        bucket = (Hash(item->key) + c1 * i + c2 * i * i) % N  
  
        // Increment number of buckets probed  
        bucketsProbed = bucketsProbed + 1  
    }  
    return false  
}
```

PARTICIPATION ACTIVITY

14.4.2: Insertion using quadratic probing.



Assume a hash function returns key % 16 and quadratic probing is used with $c1 = 1$ and $c2 = 1$.

Refer to the table below.

©zyBooks 07/15/23 19:14 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

0	32
1	49
2	16
3	3
4	
5	99
6	64
7	23
8	
9	
10	42
11	11
12	
13	
14	
15	

©zyBooks 07/15/23 19:14 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

1) 32 was inserted before 16



- True
- False

2) Which value was inserted without collision?



- 99
- 64
- 23

3) What is the probing sequence when inserting 48 into the table?



- 8
- 0, 8
- 0, 2, 6, 12

4) How many bucket index computations were necessary to insert 64 into the table?



- 1
- 2
- 3



- 5) If 21 is inserted into the hash table,
what would be the insertion index?

- 5
- 9
- 11

©zyBooks 07/15/23 19:14 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Search and removal

The search algorithm uses the probing sequence until the key being searched for is found or an empty-since-start bucket is found. The removal algorithm searches for the key to remove and, if found, marks the bucket as empty-after-removal.

PARTICIPATION ACTIVITY

14.4.3: Search and removal with quadratic probing: $c_1 = 1$ and $c_2 = 1$.



Animation captions:

1. 16, 32, and 64 all have a mapped bucket of 0. 32 was inserted first, then 16, then 64.
2. A search for 64 iterates through indices in the probe sequence: 0, 2, then 6.
3. Removal of 32 marks bucket 0 as empty-after removal.
4. A search for 64 after removing item 32 checks the empty-after-removal bucket at index 0, searches the occupied bucket at index 2, and then finds item 64 at index 6.

Figure 14.4.2: HashRemove and HashSearch with quadratic probing.

©zyBooks 07/15/23 19:14 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

```
HashRemove(hashTable, key) {
    i = 0
    bucketsProbed = 0

    // Hash function determines initial bucket
    bucket = Hash(key) % N

    while ((hashTable[bucket] is not EmptySinceStart) and (bucketsProbed <
N)) {
        if ((hashTable[bucket] is Occupied) and (hashTable[bucket]→key == key)) {
            hashTable[bucket] = EmptyAfterRemoval
            return true
        }

        // Increment i and recompute bucket index
        // c1 and c2 are programmer-defined constants for quadratic probing
        i = i + 1
        bucket = (Hash(key) + c1 * i + c2 * i * i) % N

        // Increment number of buckets probed
        bucketsProbed = bucketsProbed + 1
    }
    return false // key not found
}

HashSearch(hashTable, key) {
    i = 0
    bucketsProbed = 0

    // Hash function determines initial bucket
    bucket = Hash(key) % N

    while ((hashTable[bucket] is not EmptySinceStart) and (bucketsProbed <
N)) {
        if ((hashTable[bucket] is Occupied) and (hashTable[bucket]→key == key)) {
            return hashTable[bucket]
        }

        // Increment i and recompute bucket index
        // c1 and c2 are programmer-defined constants for quadratic probing
        i = i + 1
        bucket = (Hash(key) + c1 * i + c2 * i * i) % N

        // Increment number of buckets probed
        bucketsProbed = bucketsProbed + 1
    }
    return null // key not found
}
```

Consider the following hash table, a hash function of key % 10, and quadratic probing with $c1 = 1$ and $c2 = 1$.

valsTable:	0	1	2	3	4	5	6	7	8	9
	60									
			110							
				364						
					75					
						66				
										49

- Empty-since-start
- Empty-after-removal
- Occupied

©zyBooks 07/15/23 19:14 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 1) HashSearch(valsTable, 75) probes _____ buckets.

Check

[Show answer](#)



- 2) HashSearch(valsTable, 110) probes _____ buckets.

Check

[Show answer](#)



- 3) After removing 66 via HashRemove(valsTable, 66), HashSearch(valsTable, 66) probes _____ buckets.

Check

[Show answer](#)



©zyBooks 07/15/23 19:14 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

14.4.5: Using empty buckets during search, insertion, and removal.



- 1) When a hash table is initialized, all entries must be empty-after-removal.

True



False

- 2) The insertion algorithm can only insert into empty-since-start buckets.

 True False

- 3) The search algorithm stops only when encountering a bucket containing the key being searched for.

©zyBooks 07/15/23 19:14 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

 True False

- 4) The removal algorithm searches for the bucket containing the key to remove. If found, the bucket is marked as empty-after-removal.

 True False
CHALLENGE ACTIVITY

14.4.1: Quadratic probing.



489394.3384924.qx3zqy7

Start

valsTable:	0

 Empty-since-start Empty-after-removal Occupied

Hash table valsTable uses quadratic probing, a hash function of key % 10, c1 = 1, and c2 = 1.

HashInsert(valsTable, item 77) inserts item 77 into bucket Ex: 10

HashInsert(valsTable, item 25) inserts item 25 into bucket Taylor Larrechea

HashInsert(valsTable, item 24) inserts item 24 into bucket Summer2023

[Check](#)[Next](#)

14.5 Double hashing

©zyBooks 07/15/23 19:14 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Overview

Double hashing is an open-addressing collision resolution technique that uses 2 different hash functions to compute bucket indices. Using hash functions h_1 and h_2 , a key's index in the table is computed with the formula $i = h_1(key) + i \cdot h_2(key)$. Inserting a key uses the formula, starting with $i = 0$, to repeatedly search hash table buckets until an empty bucket is found. Each time an empty bucket is not found, i is incremented by 1. Iterating through sequential i values to obtain the desired table index is called the **probing sequence**.

PARTICIPATION ACTIVITY

14.5.1: Hash table insertion using double hashing.



Animation captions:

1. Items 72, 60, 45, 18, and 39 are inserted without collisions.
2. When inserting item 55, bucket 5 is occupied. Incrementing i to 1 and recomputing the hash function yields an empty bucket at index 3 for item 55.
3. Inserting item 23 also result in collisions. i is incremented to 2 before finding an empty bucket at index 10 to insert item 23.

PARTICIPATION ACTIVITY

14.5.2: Double hashing.



Given:

$$\text{hash1(key)} = \text{key \% 11}$$

$$\text{hash2(key)} = 5 - \text{key \% 5}$$

and a hash table with a size of 11. Determine the index for each item after the following operations have been executed.

©zyBooks 07/15/23 19:14 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

```
HashInsert(valsTable, item 16)
HashInsert(valsTable, item 77)
HashInsert(valsTable, item 55)
HashInsert(valsTable, item 41)
HashInsert(valsTable, item 63)
```

1) Item 16

//
Check**Show answer**

2) Item 55

//
Check**Show answer**

©zyBooks 07/15/23 19:14 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023



3) Item 63

//
Check**Show answer**

Insertion, search, and removal

Using double hashing, a hash table search algorithm probes (or checks) each bucket using the probing sequence defined by the two hash functions. The search continues until either the matching item is found (returning the item), an empty-since-start bucket is found (returning null), or all buckets are probed without a match (returning null).

A hash table insert algorithm probes each bucket using the probing sequence, and inserts the item in the next empty bucket (the empty kind doesn't matter).

A hash table removal algorithm first searches for the item's key. If the item is found, the item is removed, and the bucket is marked empty-after-removal.

PARTICIPATION ACTIVITY

14.5.3: Hash table insertion, search, and removal using double hashing.



Animation captions:

1. When item 3 is removed, the bucket is marked as empty-after-removal.
2. Search for 19 checks bucket 3 first. The bucket is empty-after-removal, and additional buckets must be searched.
3. Inserting item 88 has a collision at bucket 8. The next bucket index of 3 yields an empty-after-removal bucket, and item 88 is inserted in that bucket.

PARTICIPATION ACTIVITY

14.5.4: Hash table with double hashing: search, insert, and remove.



Consider the following hash table, a first hash function of key % 10, and a second hash function of 7 - key % 7.

valsTable:	0	1	2	3	4	5	6	7	8	9
	60									
				223						
					104					
				66						
							49			

- Empty-since-start
- Empty-after-removal
- Occupied

©zyBooks 07/15/23 19:14 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 1) HashSearch(valsTable, 110) probes _____ buckets.

Check

[Show answer](#)



- 2) HashInsert(valsTable, item 24) probes _____ buckets.

Check

[Show answer](#)



- 3) After removing 66 via HashRemove(valsTable, 66), HashSearch(valsTable, 66) probes _____ buckets.

Check

[Show answer](#)



PARTICIPATION ACTIVITY

14.5.5: Hash table insertion, search, and removal with double hashing.

©zyBooks 07/15/23 19:14 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 1) When the removal algorithm finds the bucket containing the key to be removed, the bucket is marked as empty-since-start.



- True
- False

2) Double hashing would never resolve collisions if the second hash function always returned 0.

- True
- False

©zyBooks 07/15/23 19:14 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

**CHALLENGE ACTIVITY****14.5.1: Double hashing.**

489394.3384924.qx3zqy7

Start

valsTable:	0
	1
	2
	3
	4 48
	5
	6
	7
	8
	9 86
	10

- Empty-since-start
- Empty-after-removal
- Occupied

Hash table valsTable uses double probing with the hash functions
 $\text{hash1(key)}: \text{key \% 11}$
 $\text{hash2(key)}: 5 - \text{key \% 5}$
and a table size of 11.

HashInsert(valsTable, item 13) inserts item 13 into bucket Ex: 10

HashInsert(valsTable, item 26) inserts item 26 into bucket

HashInsert(valsTable, item 70) inserts item 70 into bucket

1

2

3

4

Check**Next**

©zyBooks 07/15/23 19:14 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

14.6 Hash table resizing

Resize operation

A hash table **resize** operation increases the number of buckets, while preserving all existing items. A hash table with N buckets is commonly resized to the next prime number $\geq N * 2$. A new array is allocated, and all items from the old array are re-inserted into the new array, making the resize operation's time complexity

PARTICIPATION
ACTIVITY

14.6.1: Hash table resize operation.

©zyBooks 07/15/23 19:14 1692462

Taylor Larrechea

COLORADO CSPB2270Summer2023



Animation content:

undefined

Animation captions:

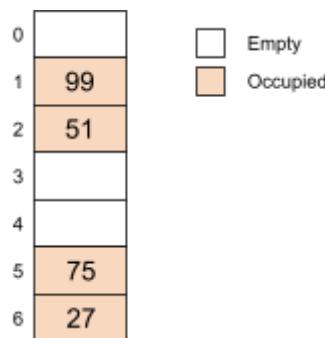
1. When resizing a hash table with 11 buckets and 7 items, the new size is computed as the next prime number ≥ 22 , which is 23.
2. A new array is allocated with 23 buckets for the resized hash table.
3. When rehashing 88, the bucket index is computed as $88 \% 23 = 19$. newArray[19] is assigned with 88.
4. The key from each of hashTable's non-empty buckets is rehashed and inserted into newArray.
5. newArray is returned and is the resized hash table.

PARTICIPATION
ACTIVITY

14.6.2: Resizing a hash table.



Suppose the hash table below is resized. The hash function used both before and after resizing is: $\text{hash(key)} = \text{key \% } N$, where N is the table size.



©zyBooks 07/15/23 19:14 1692462

Taylor Larrechea

COLORADO CSPB2270Summer2023

- 1) What is the most likely allocated size for the resized hash table?

- 7
- 14
- 17





- 2) How many elements are in the hash table after resizing?

- 0
- 4
- 7

- 3) At what index does 99 reside in the resized table?

- 1
- 9
- 14

©zyBooks 07/15/23 19:14 169246
Taylor Larrechea
COLORADOCSPB2270Summer2023

When to resize

A hash table's **load factor** is the number of items in the hash table divided by the number of buckets.
Ex: A hash table with 18 items and 31 buckets has a load factor of $\frac{18}{31}$. The load factor may be used to decide when to resize the hash table.

An implementation may choose to resize the hash table when one or more of the following values exceeds a certain threshold:

- Load factor
- When using open-addressing, the number of collisions during an insertion
- When using chaining, the size of a bucket's linked-list

PARTICIPATION ACTIVITY

14.6.3: Resizing when a chaining bucket is too large.



Animation captions:

1. A hash table with chaining will inevitably have large linked-lists after inserting many items.
2. The largest bucket length can be used as resizing criteria. Ex: The hash table is resized when a bucket length is ≥ 4 .

PARTICIPATION ACTIVITY

14.6.4: Resizing when the load factor is ≥ 0.6 .

©zyBooks 07/15/23 19:14 169246
Taylor Larrechea
COLORADOCSPB2270Summer2023



Animation captions:

1. A hash table with 2 items and 5 buckets has a load factor of $2 / 5 = 0.4$.
2. Inserting 22 increases the load factor to $3 / 5 = 0.6$.
3. An implementation may choose to resize the hash table whenever the load factor is ≥ 0.6 .

PARTICIPATION ACTIVITY

14.6.5: Resizing when an insertion causes more than N / 3 collisions.

**Animation captions:**

1. Inserting 38 into the hash table with linear probing encounters 5 collisions before placing 38 in bucket 10.
©zyBooks 07/15/23 19:14 1692462
Taylor Larrechea
2. If the hash table's resize criteria were to resize after encountering ADOCSPB2270 collisions, then the insertion would cause a resize.

PARTICIPATION ACTIVITY

14.6.6: Resize criteria and load factors.



- 1) A hash table implementation must use only one criteria for resizing.
 True
 False
- 2) In a hash table using open addressing, the load factor cannot exceed 1.0.
 True
 False
- 3) In a hash table using chaining, the load factor cannot exceed 1.0.
 True
 False
- 4) When resizing to a larger size, the load factor is guaranteed to decrease.
 True
 False

PARTICIPATION ACTIVITY

14.6.7: Resizing a hash table with 101 buckets.



Suppose a hash table has 101 buckets.

- 1) If the hash table was using chaining, the load factor could be ≤ 0.1 , but an

individual bucket could still contain 10 items.

- True
- False

2) If the hash table was using open addressing, a load factor < 0.25 guarantees that no more than 25 collisions will occur during insertion.

- True
- False

3) If the hash table was using open addressing, a load factor > 0.9 guarantees a collision during insertion.

- True
- False

©zyBooks 07/15/23 19:14 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



14.7 Common hash functions

A good hash function minimizes collisions

A hash table is fast if the hash function minimizes collisions.

A **perfect hash function** maps items to buckets with no collisions. A perfect hash function can be created if the number of items and all possible item keys are known beforehand. The runtime for insert, search, and remove is $O(1)$ with a perfect hash function.

A good hash function should uniformly distribute items into buckets. With chaining, a good hash function results in short bucket lists and thus fast inserts, searches, and removes. With linear probing, a good hash function will avoid hashing multiple items to consecutive buckets and thus minimize the average linear probing length to achieve fast inserts, searches, and removes. On average, a good hash function will achieve $O(1)$ inserts, searches, and removes, but in the worst-case may require $O(N)$.

A hash function's performance depends on the hash table size and knowledge of the expected keys. Ex: The hash function key \% 10 will perform poorly if the expected keys are all multiples of 10, because inserting 10, 20, 30, ..., 90, and 100 will all collide at bucket 0.

Modulo hash function

A **modulo hash** uses the remainder from division of the key by hash table size N.

Figure 14.7.1: Modulo hash function.

```
HashRemainder(int key)
{
    return key % N
}
```

©zyBooks 07/15/23 19:14 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

PARTICIPATION
ACTIVITY

14.7.1: Good hash functions and keys.



Will the hash function and expected key likely work well for the following scenarios?

1) Hash function: key % 1000



Key: 6-digit employee ID

Hash table size: 20000

- Yes
- No

2) Hash function: key % 250



Key: 5-digit customer ID

Hash table size: 250

- Yes
- No

3) Hash function: key % 1000



Key: Selling price of a house.

Hash table size: 1000

- Yes
- No

4) Hash function: key % 40

©zyBooks 07/15/23 19:14 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



Key: 4-digit even numbers

Hash table size: 40

- Yes
- No



5) Hash function: key % 1000

Key: Customer's 3-digit U.S. phone number area code, of which about 300 exist.

Hash table size: 1000

Yes

No

©zyBooks 07/15/23 19:14 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Mid-square hash function

A **mid-square hash** squares the key, extracts R digits from the result's middle, and returns the remainder of the middle digits divided by hash table size N. Ex: For a hash table with 100 entries and a key of 453, the decimal (base 10) mid-square hash function computes $453 * 453 = 205209$, and returns the middle two digits 52. For N buckets, R must be greater than or equal to _____ to index all buckets. The process of squaring and extracting middle digits reduces the likelihood of keys mapping to just a few buckets.

PARTICIPATION ACTIVITY

14.7.2: Decimal mid-square hash function.



- 1) For a decimal mid-square hash function, what are the middle digits for key = 40, N = 100, and R = 2?

Check

[Show answer](#)



- 2) For a decimal mid-square hash function, what is the bucket index for key = 110, N = 200, and R = 3?

Check

[Show answer](#)



- 3) For a decimal mid-square hash function, what is the bucket index for key = 112, N = 1000, and R = 3?

Check

[Show answer](#)

©zyBooks 07/15/23 19:14 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



Mid-square hash function base 2 implementation

The mid-square hash function is typically implemented using binary (base 2), and not decimal, because a binary implementation is faster. A decimal implementation requires converting the square of the key to a string, extracting a substring for the middle digits, and converting that substring to an integer. A binary implementation only requires a few shift and bitwise AND operations.

A binary mid-square hash function extracts the middle R bits, and returns the remainder of the middle bits divided by hash table size N, where R is greater than or equal to . Ex: For a hash table size of 200, R = 8, then 8 bits are needed for indices 0 to 199.

Figure 14.7.2: Mid-square hash function (base 2).

```
HashMidSquare(int key) {
    squaredKey = key * key

    lowBitsToRemove = (32 - R) / 2
    extractedBits = squaredKey >> lowBitsToRemove
    extractedBits = extractedBits & (0xFFFFFFFF >> (32 -
R))

    return extractedBits % N
}
```

The extracted middle bits depend on the maximum key. Ex: A key with a value of 4000 requires 12 bits. A 12-bit number squared requires up to 24 bits. For R = 10, the middle 10 bits of the 24-bit squared key are bits 7 to 16.

PARTICIPATION ACTIVITY

14.7.3: Binary mid-square hash function.



- 1) For a binary mid-square hash function, how many bits are needed for an 80 entry hash table?

Check

Show answer

©zyBooks 07/15/23 19:14 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 2) For R = 3, what are the middle bits for a key of 9? $9 * 9 = 81$; 81 in binary is 1010001.



 Check Show answer

©zyBooks 07/15/23 19:14 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Multiplicative string hash function

A **multiplicative string hash** repeatedly multiplies the hash value and adds the ASCII (or Unicode) value of each character in the string. A multiplicative hash function for strings starts with a large initial value. For each character, the hash function multiplies the current hash value by a multiplier (often prime) and adds the character's value. Finally, the function returns the remainder of the sum divided by the hash table size N.

Figure 14.7.3: Multiplicative string hash function.

```
HashMultiplicative(string key) {  
    stringHash = initialValue  
  
    for (each character strChar in key) {  
        stringHash = (stringHash * HashMultiplier) +  
strChar  
    }  
  
    return stringHash % N  
}
```

Daniel J. Bernstein created a popular version of a multiplicative string hash function that uses an initial value of 5381 and a multiplier of 33. Bernstein's hash function performs well for hashing short English strings.

PARTICIPATION ACTIVITY

14.7.4: Multiplicative string hash function.

©zyBooks 07/15/23 19:14 1692462

Taylor Larrechea
COLORADOCSPB2270Summer2023

For a 1000-entry hash table, compute the multiplicative hash for the following strings using the specific initial value and hash multiplier. The decimal ASCII value for each character is shown below.

Character	Decimal value	Character	Decimal value
A	65	N	78

B	66	O	79
C	67	P	80
D	68	Q	81
E	69	R	82
F	70	S	83
G	71	T	84
H	72	U	85
I	73	V	86
J	74	W	87
K	75	X	88
L	76	Y	89
M	77	Z	90

©zyBooks 07/15/23 19:14 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

- 1) Initial value = 0

Hash multiplier = 1

String = BAT



//

Check

Show answer

- 2) Initial value = 0

Hash multiplier = 1

String = TAB



//

Check

Show answer

- 3) Initial value = 17

Hash multiplier = 3

String = WE



©zyBooks 07/15/23 19:14 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

Check**Show answer**

Exploring further:

The following provide resources that summarize, discuss, and analyze numerous hash functions.

©zyBooks 07/15/23 19:14 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

- [Hash Functions: An Empirical Comparison](#) by Peter Kankowski
- [Hash Functions and Block Ciphers](#) by Bob Jenkins.

14.8 Direct hashing

Direct hashing overview

A **direct hash function** uses the item's key as the bucket index. Ex: If the key is 937, the index is 937. A hash table with a direct hash function is called a **direct access table**. Given a key, a direct access table **search** algorithm returns the item at index key if the bucket is not empty, and returns null (indicating item not found) if empty.

PARTICIPATION ACTIVITY

14.8.1: Direct hash function.



Animation content:

undefined

Animation captions:

1. A direct hash function uses the item's key as the bucket index. The value stored in hashTable[6] is returned.
2. hashTable[3] is empty. Search returns null, indicating the item was not found.

©zyBooks 07/15/23 19:14 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Figure 14.8.1: Direct hashing: Insert, remove, and search operations use item's key as bucket index.

```

HashInsert(hashTable, item) {
    hashTable[item->key] = item
}

HashRemove(hashTable, item) {
    hashTable[item->key] = Empty
}

HashSearch(hashTable, key) {
    if (hashTable[key] is not Empty)
    {
        return hashTable[key]
    }
    else {
        return null
    }
}

```

©zyBooks 07/15/23 19:14 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

**PARTICIPATION
ACTIVITY**
14.8.2: Direct access table search, insert, and remove.


Type the hash table after the given operations. Type the hash table as: E, 1, 2, E, E (where E means empty).

1)

numsTable:	0	
	1	1
	2	2
	3	
	4	



HashInsert(numsTable, item 0)

numsTable:

Check
Show answer

2)

numsTable:	0	0
	1	
	2	2
	3	3
	4	



HashRemove(numsTable, 0)

HashInsert(numsTable, item 4)

numsTable:


©zyBooks 07/15/23 19:14 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Check**Show answer**

Limitations of direct hashing

A direct access table has the advantage of no collisions: Each key is unique (by definition of a key), and each gets a unique bucket, so no collisions can occur. However, a direct access table has two main limitations.

©zyBooks 07/15/23 19:14 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

1. All keys must be non-negative integers, but for some applications keys may be negative.
2. The hash table's size equals the largest key value plus 1, which may be very large.

PARTICIPATION ACTIVITY

14.8.3: Direct hashing limitations.



- 1) For a 1000-entry direct access table, type the bucket number for the inserted item, or type: None



HashInsert(hashIndex, item 734)

 //**Check****Show answer**

- 2) For a 1000-entry direct access table, type the bucket number for the inserted item, or type: None



HashInsert(hashIndex, item 1034)

 //**Check****Show answer**

- 3) For a 1000-entry direct access table, type the bucket number for the inserted item, or type: None

©zyBooks 07/15/23 19:14 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

HashInsert(hashIndex, item -45)

 //**Check****Show answer**



- 4) How many direct access table buckets are needed for items with keys ranging from 100 to 200 (inclusive)?

 //**Check****Show answer**

©zyBooks 07/15/23 19:14 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



- 5) A class has 100 students. Student ID numbers range from 10000 to 99999. Using the ID number as key, how many buckets will a direct access table require?

 //**Check****Show answer**

14.9 Hashing Algorithms: Cryptography, Password Hashing

Cryptography

Cryptography is a field of study focused on transmitting data securely. Secure data transmission commonly starts with **encryption**: alteration of data to hide the original meaning. The counterpart to encryption is **decryption**: reconstruction of original data from encrypted data.

PARTICIPATION ACTIVITY

14.9.1: Basic encryption: Caesar cipher.



Animation captions:

©zyBooks 07/15/23 19:14 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

1. The Caesar cipher shifts characters in the alphabet to encrypt a message. With a right shift of 4, the character 'N' is shifted to 'R'.
2. The shift is applied to each character in the string, including spaces. The result is an encrypted message that hides the original message.
3. A left shift can also be used.
4. Each message can be decrypted with the opposite shift.

PARTICIPATION ACTIVITY

14.9.2: Caesar cipher.



1) What is the result of applying the Caesar cipher with a left shift of 1 to the string "computer"?

- eqorwvgt
- dpnqvufs
- bnlotsdq

©zyBooks 07/15/23 19:14 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

2) If a message is encrypted with a left shift of X, what shift is needed to decrypt?

- left shift of X
- right shift of X



3) If the Caesar cipher were implemented such that strings were restricted to only lower-case alphabet characters, how many distinct ways could a message be encrypted?

- 25
- 50
- Length of the message

**PARTICIPATION ACTIVITY**

14.9.3: Cryptography.



1) Encryption and decryption are synonymous.

- True
- False



2) Cryptography is used heavily in internet communications.

- True
- False

©zyBooks 07/15/23 19:14 1692462

Taylor Larrechea



COLORADOCSPB2270Summer2023

3) The Caesar cipher is an encryption algorithm that works well to secure



data for modern digital communications.

- True
- False

©zyBooks 07/15/23 19:14 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Hashing functions for data

A hash function can be used to produce a hash value for data in contexts other than inserting the data into a hash table. Such a function is commonly used for the purpose of verifying data integrity. Ex: A hashing algorithm called MD5 produces a 128-bit hash value for any input data. The hash value cannot be used to reconstruct the original data, but can be used to help verify that data isn't corrupt and hasn't been altered.

PARTICIPATION
ACTIVITY

14.9.4: A hash value can help identify corrupted data downloaded from the internet.



Animation captions:

1. Computer B will attempt to download a message over the internet from computer A. Computer B will also download a corresponding 128-bit MD5 hash value from computer A.
2. Due to an unreliable network, the message data arrives corrupted. The MD5 hash is downloaded correctly.
3. Computer B computes the MD5 hash for the downloaded data. The computed hash is different from the downloaded hash, implying the data was corrupted.

PARTICIPATION
ACTIVITY

14.9.5: Hashing functions for data.



- 1) MD5 produces larger hash values for larger input data sizes.

- True
- False

©zyBooks 07/15/23 19:14 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

- 2) A hash value can be used to reconstruct the original data.

- True
- False





3) If computer B in the above example computed a hash value identical to the downloaded hash value, then the downloaded message would be guaranteed to be uncorrupted.

- True
- False

©zyBooks 07/15/23 19:14 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Cryptographic hashing

A **cryptographic hash function** is a hash function designed specifically for cryptography. Such a function is commonly used for encrypting and decrypting data.

A **password hashing function** is a cryptographic hashing function that produces a hash value for a password. Databases for online services commonly store a user's password hash as opposed to the actual password. When the user attempts a login, the supplied password is hashed, and the hash is compared against the database's hash value. Because the passwords are not stored, if a database with password hashes is breached, attackers may still have a difficult time determining a user's password.

PARTICIPATION ACTIVITY

14.9.6: Password hashing function.



Animation captions:

1. A password hashing function produces a hash value for a password.
2. The password hash function aims to produce a hash that cannot easily be converted back to the password.
3. Also, two different passwords should not produce the same hash value.
4. Some password hashing functions concatenate extra random data to the password, then store the random data as well as the password hash value.

PARTICIPATION ACTIVITY

14.9.7: Password hashing function.



1) Which is not an advantage of storing password hash values, instead of actual passwords, in a database?

- Database administrators cannot see users' passwords.
- Database storage space is saved.

©zyBooks 07/15/23 19:14 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

- Attackers who gain access to database contents still may not be able to determine users' passwords.

©zyBooks 07/15/23 19:14 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 2) A user could login with an incorrect password if a password hashing function produced the same hash value for two different passwords.

- True
 False



- 3) Generating and storing random data alongside each password hash in a database, and using (password + random_data) to generate the hash value, can help increase security.

- True
 False



©zyBooks 07/15/23 19:14 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

15.1 Compression

Compression basics

Compression reduces the size of a file. Ex: An image/photo file may be 4 megabytes before compression, but only 1 megabyte after.

Compression enables a user to store more files on a drive. Compression also speeds up transfers of files via the Internet.

Compression can be applied to any file, but is commonly applied to audio, image, and video files because such files are naturally large. Most apps for such files automatically compress and decompress those files.

PARTICIPATION ACTIVITY

15.1.1: Compression reduces file size, enabling storage of more files, and faster file transfers.



Animation captions:

1. Compression reduces the size of a file.
2. Compression enables more files to be stored on a drive.
3. Compression also speeds up file transfers, like web downloads.

PARTICIPATION ACTIVITY

15.1.2: Compression basics.



- 1) By reducing a file's size, compression enables more files to be stored on a drive.

- True
 False

- 2) Compression has the drawback of increasing the time to download a file via the web.

- True
 False

©zyBooks 07/17/23 16:58 169246
Taylor Larrechea
COLORADOCSPB2270Summer2023

Dictionary-based compression

A common approach to compressing data in a file is to create a custom dictionary for that data. In compression, a **dictionary** is a table of shorthand versions of longer data. Ex: Given a dictionary of 1: Department, 2: of, and 3: Redundancy (1, 2, and 3 are the shorthand items), then "1 2 3 1" is short for "Department of Redundancy Department". The compressed file will contain the dictionary itself plus the data in shorthand form, which combined is hopefully smaller than the original file. A typical text file might be compressed by 50% or more.

An **LZ compression** algorithm (named for creators Lempel and Ziv) examines data for long repeating patterns such as phrases, and creates a dictionary entry for such patterns. Alternatively, a **Huffman encoding** algorithm measures the frequency of each data item like each letter, and gives the most frequent items a shorter bit encoding (like the letters "a" and "e"), with least frequent items getting a longer encoding (like letters "q" and "z"). Many compression techniques combine LZ and Huffman algorithms.

PARTICIPATION ACTIVITY

15.1.3: Compression using an LZ algorithm dictionary approach.

**Animation captions:**

1. LZ compression algorithms search for repeating patterns, creating a dictionary entry for each.
2. The compressed file contains the dictionary, and the data in shorthand.
3. The compressed file, having both the dictionary and the shorthand data, may be smaller.
Compressions of 50%-90% are common.

PARTICIPATION ACTIVITY

15.1.4: Dictionary-based compression.



- 1) Given the dictionary below for an LZ algorithm, what uncompressed phrase do these numbers represent: 1 3 2



- 1: Oh
2: gosh
3: my
- Oh gosh my
 - Oh my gosh
 - Cannot be determined

©zyBooks 07/17/23 16:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 2) Given the following uncompressed text, which is a more reasonable LZ dictionary?



He sells sea shells.

1: ellipsis

1: e

2: s

- 3) How much smaller is the compressed text than the original? Use this equation: (uncompressed - compressed) / uncompressed. Count each character, including spaces.

©zyBooks 07/17/23 16:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Original: To infinity and beyond!

Compressed: To 1 and 2!

52%

48%

0%

- 4) Given the below Huffman coding dictionary, what uncompressed letters do these bits represent: 01 01 01 001 00011111 01



01: a

001: b

00011111: z

a b z a

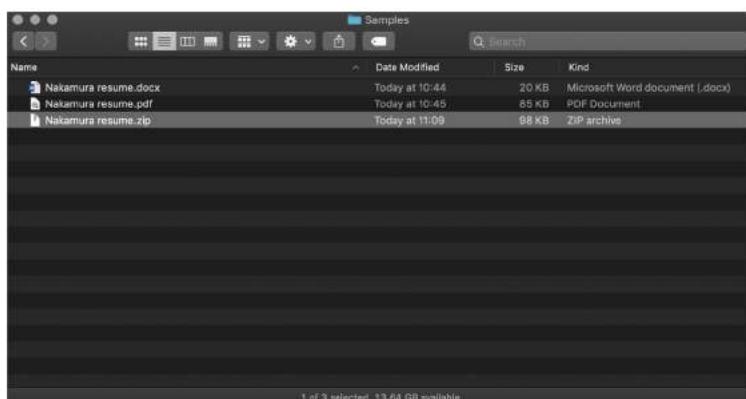
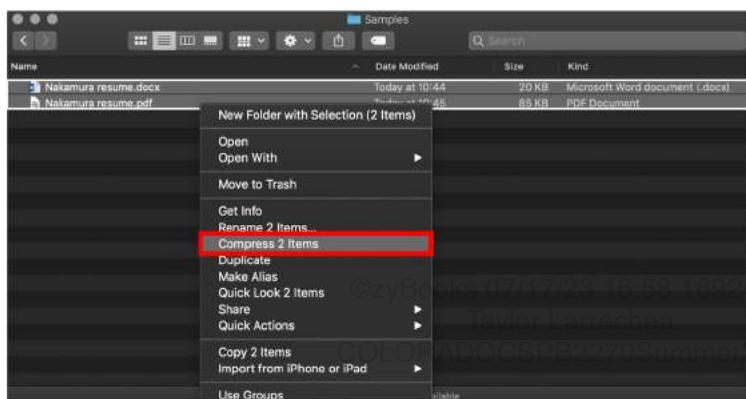
a a a b z a

Zip files

A **zip file** is a common file format for combining multiple files into one file and that usually involves compression. Ex: Using a zip app on a Mac, a 20 kbyte Microsoft Word file and a 85 kbyte PDF file are compressed into a single 98 kbyte zip file..

Taylor Larrechea
COLORADOCSPB2270Summer2023

Figure 15.1.1: File compression/uncompression on a Mac computer.



Compress

1. Right-click on file(s)
2. Select “Compress”
3. Compressed file appears in folder

Uncompress

1. Double-click on zip file
2. Uncompressed file appears in folder

Figure 15.1.2: File compression on a Windows computer.

Compress

1. Right-click on file
2. Select "Send To"
3. Select "Compressed (zipped) Folder"
4. Compressed file appears in folder

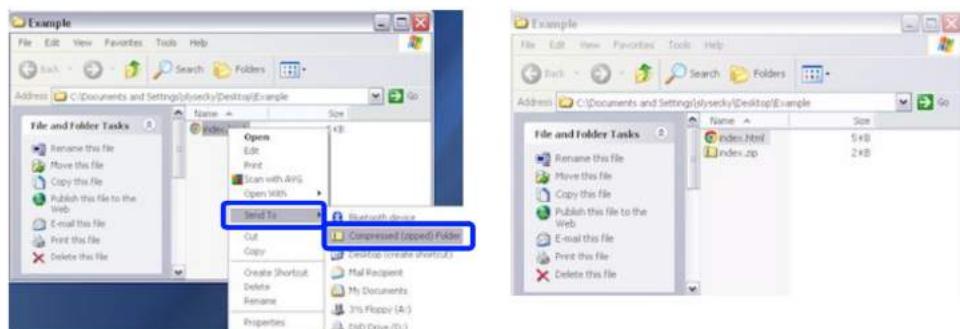


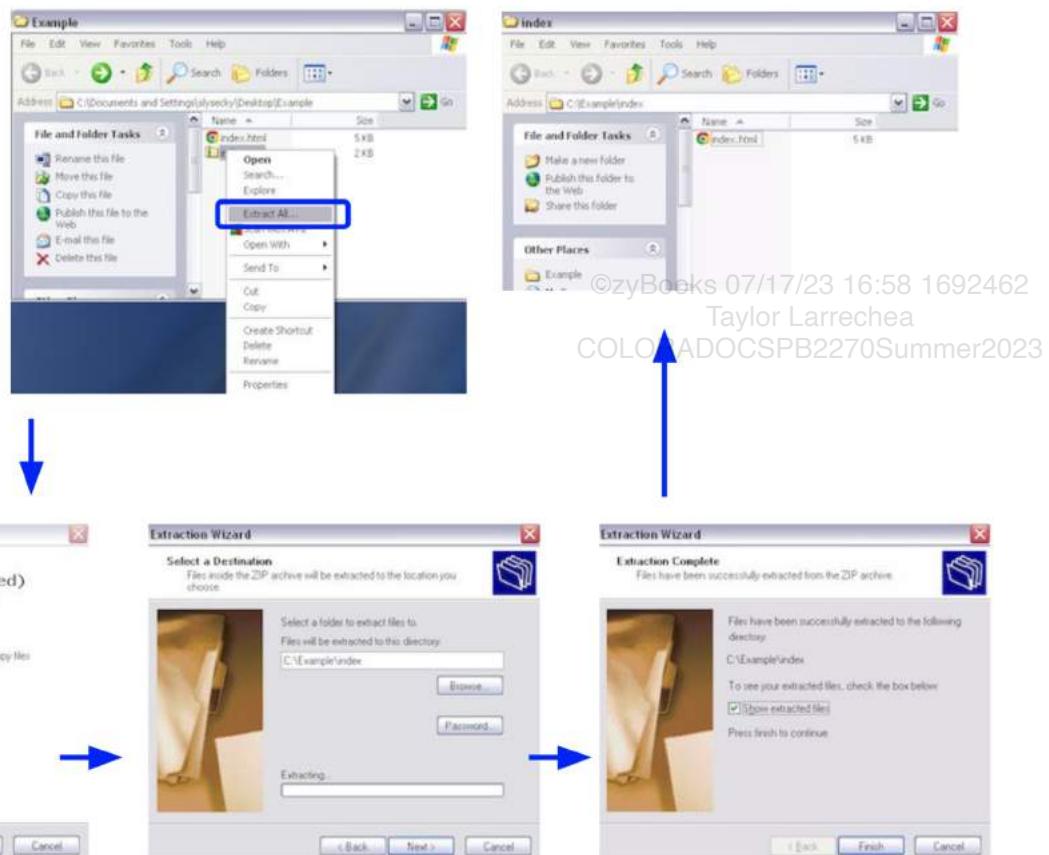
Figure 15.1.3: Uncompressing a file on a Windows computer.

7/17/23 16:58 1692462

Taylor Larrechea
COLORADOCSPB2270Summer2023

Uncompress

1. Right-click on zip file
2. Select "Extract All..."
3. Select "Next"
4. Specify file destination
5. Select "Finish"
6. Uncompressed file appears in folder

**PARTICIPATION ACTIVITY****15.1.5: Zip files.**

- 1) A zip file is commonly created to collect multiple files into a single file.

- True
 False

- 2) The process of creating a zip file may compress the file's contents.

- True
 False

Lossless versus lossy compression

Lossless compression loses no information, so that decompression yields an identical file to the original. LZ and Huffman approaches are lossless.

Lossy compression loses some information, so the decompressed file is close but not identical to the original. An example lossy compression approach is rounding. Ex: Given an original file containing 255, 64, 231, the one's place can be dropped (a form of rounding), yielding 25, 6, 23. A decompressor,

©zyBooks 07/17/23 16:58 1692462

Taylor Larrechea
COLORADOCSPB2270Summer2023

knowing of the rounding but not knowing what numbers were dropped, may append 5's, yielding 255, 65, and 235, which is close but not identical to the original. Lossy compression may be OK for data like images or audio where humans barely notice a quality difference, but is clearly not OK for precise data like text or bank account numbers since the meaning would change.

PARTICIPATION ACTIVITY**15.1.6: Lossless versus lossy compression.**

©zyBooks 07/17/23 16:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

- 1) Lossy compression means that some files that were to be compressed were lost.

- True
 False

- 2) Lossy compression is acceptable for a file containing music for casual listening.

- True
 False

- 3) Lossy compression is acceptable for a file containing phone numbers.

- True
 False

- 4) Compression is not possible unless loss of information is acceptable.

- True
 False

JPEG compression

JPEG is a compression approach specifically for images. An image may consist of millions of **pixels** (short for "picture elements"), each pixel being a colored dot. A pixel may be 3 numbers (each a byte) indicating the amount of red, green, and blue. If an image has 4 million pixels, and each pixel requires 3 bytes, then a single uncompressed image is basically just a series of 12 million numbers (so 12 Mbytes). JPEG compresses the image using several techniques.

- One is to convert to the "frequency domain", which is beyond this material's scope.
- A second is to use Huffman encoding. Ex: An image with a lot of bright pixels may have millions of "255" values for pixel colors. 255 in binary is usually 11111111, but Huffman encoding may

create a dictionary entry like 01: 11111111, so those millions of 11111111's can be replaced by 01's.

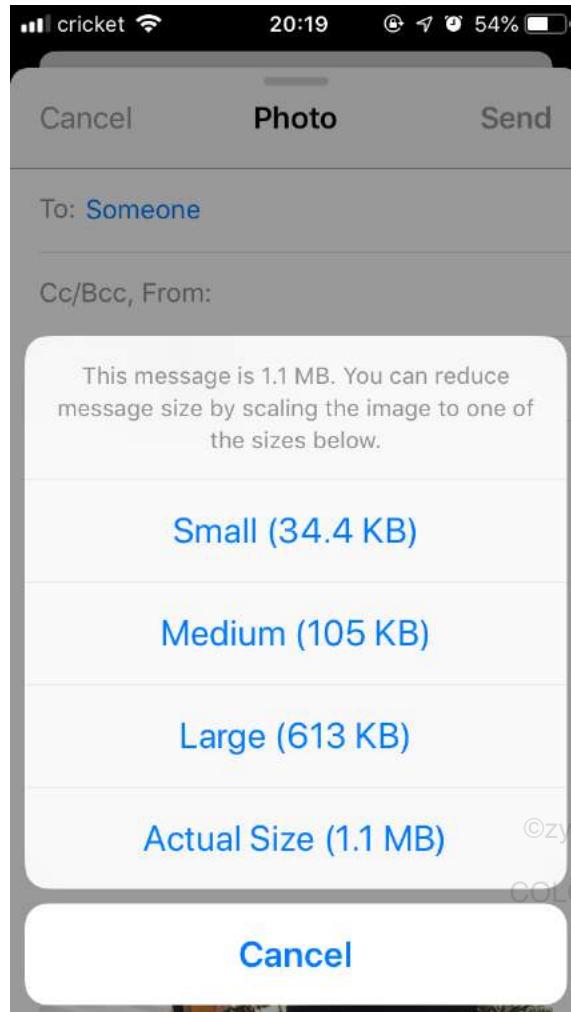
- A third compression approach is to round the numbers, known as **quantization**. So a pixel of 255 red, 64 green, and 231 blue (which is a shade of purple) may become 25, 6, and 23. Decompression might append a 5, yielding 255, 65, and 235, which is an unnoticeably different purple than the original.

(Note: The conversion to frequency domain would modify the discussion of the latter two steps, but the intuition is the same.)

©zyBooks 07/17/23 16:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

JPEG is lossy compression, due in part to the rounding (as well as conversion to frequency domain). Image apps may allow a user to reduce file size, achieved by losing more information, such as by doing even more rounding. The loss in quality may not be noticeable unless the image is enlarged.

Figure 15.1.4: iPhone asks a user to choose the size of the photo transmitted.



©zyBooks 07/17/23 16:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Figure 15.1.5: Mac Preview (and other computer apps) enable a user to resize an image.

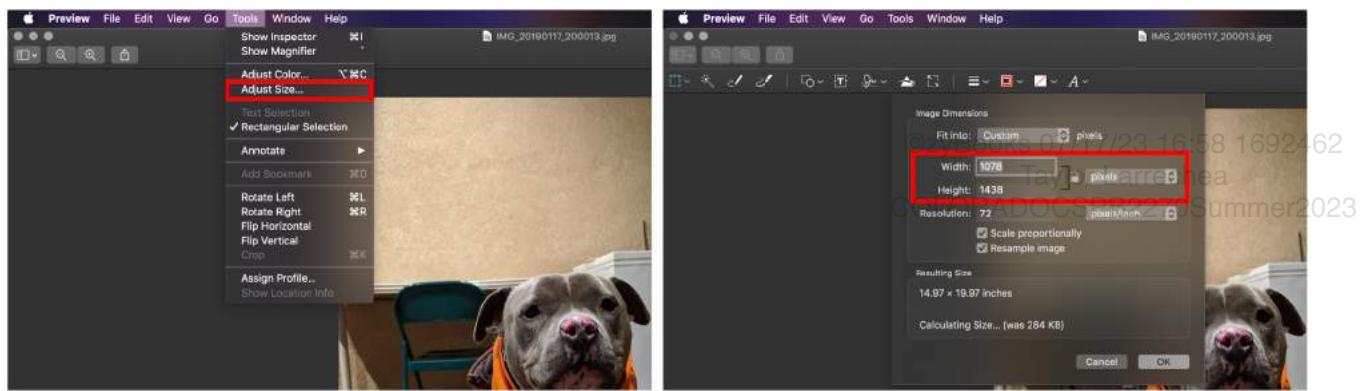


Figure 15.1.6: For a highly-compressed JPEG image, a small image displays well, but the image enlarged shows the loss in quality.

©zyBooks 07/17/23 16:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Smaller image:



©zyBooks 07/17/23 16:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Enlarging the image leads to pixelation:



**PARTICIPATION
ACTIVITY**

15.1.7: JPEG compression.



- 1) JPEG is a ____ compression approach for images.



- lossless
- lossy

- 2) ____ is a compression approach where pixel values are rounded. Ex: 145 is rounded to 14 by dropping the one's place.



- Huffman encoding
- JPEG
- Quantization

©zyBooks 07/17/23 16:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



- 3) Given the number of occurrences of a pixel color, which Huffman encoding yields the best compression?

Pixel value	# of occurrences
11111111	5,200,000
11110000	4,300,000
00000011	1,270,000

©zyBooks 07/17/23 16:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

- 011: 11111111
101: 11110000
111: 00000011
- 01: 11111111
001: 11110000
0001: 00000011

Video compression

MPEG is a compression approach specifically for video (a recent well-known version of MPEG is MP4).

Video is a series of images called **frames**. If shown faster than about 15 frames per second (movies use 24, TVs use 30 or more), a human's vision system sees the images as a continuous video. The key idea of video compression is that successive frames differ only slightly, so a frame can be represented just as the difference from the previous frame. Ex: Frame 1 may be a full image, but Frame 2 may just be "Previous frame shifted left 2 pixels".

Clearly not all frames can be represented as the difference of another frame. Thus, such video compression sends an image frame and then perhaps 10 "predicted" frames, followed by another image frame. Note: Image frames are also compressed using image compression like JPEG.

H264 is a more recent video compression approach than MPEG, intended to reduce bits further for fast transmission of video over networks.

Video compression is lossy, in part due to the images being compressed using JPEG (which is lossy), and more so because predicted frames clearly aren't entirely accurate. Video apps may support different quality levels, with lower quality achieving smaller file sizes via use of more predicted frames and more-compression of images too. Video compression amounts of 50x-100x or more are common.

©zyBooks 07/17/23 16:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

15.1.8: Video compression.





1) A video consists of a woman sitting and presenting the news. Is this video amenable to extensive compression?

- Yes
- No

2) A video consists of a car exploding. Is this video amenable to extensive compression?

©zyBooks 07/17/23 16:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- Yes
- No

3) To compress video, an app reduces frames per second to eight. Will the video quality be acceptable?

- Yes
- No

4) Was H264 created before MPEG?

- Yes
- No



Audio compression

MP3 and **OGG** are audio compression techniques. Audio is captured electronically as varying voltages on a wire. Those voltages are converted to numbers for digital storage. An uncompressed 3-minute song may require tens of megabytes. A **WAV** file stores audio uncompressed. MP3 and OGG compression use techniques (introduced above) like converting to frequency domain, quantization, and Huffman coding. A 3-minute song may be compressed to just a few megabytes.

MP3's name comes from MPEG, as MP3 was used for compressing the audio part of video files. OGG was developed as a free open-source audio compression technique.

PARTICIPATION ACTIVITY

15.1.9: Compression for video and audio files.

©zyBooks 07/17/23 16:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



If unable to drag and drop, refresh the page.

MP3

WAV

H264

Video compression technique.

Audio compression technique.

Uncompressed audio file.

Reset

©zyBooks 07/17/23 16:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Exploring further:

- [Data compression \(Wikipedia\)](#).
- [How file compression works \(How stuff works\)](#).

15.2 Data compression

Basic compression idea

Given data represented as some quantity of bits, **compression** transforms the data to use fewer bits. Compressed data uses less storage, and can be communicated faster too.

The basic idea of compression is to encode frequently-occurring items using fewer bits. Ex: ASCII characters use 8 bits each, but instead more-frequently-occurring characters could use fewer bits and other characters use more bits.

PARTICIPATION
ACTIVITY

15.2.1: The basic ideas of compression is to use fewer bits for frequent items (and more bits for less-frequent items).



Animation captions:

1. The text "AAA Go" as ASCII would use $6 * 8 = 48$ bits. Such data is uncompressed.
2. Compression uses a dictionary of codes specifically for the data. Frequent items get shorter codes. Here, A (which is most frequent) is 0, space 10, G 110, and o 111.
3. Thus, "AAA Go" is compressed as 0 0 0 10 110 111. The compressed data uses only eleven bits, much fewer than the 48 bits uncompressed.

The example above has only four distinct characters (A, space, G, and o) so could be encoded using 2 bits (fixed-length code). However, the example is trivially simple, for learning. Actual text may use all

ASCII characters so a fixed-length code would require 8 bits per character, but with varying-length codes as above where the frequent characters in the data might use fewer bits (like 4).

PARTICIPATION ACTIVITY**15.2.2: Basic compression.**

Given the following dictionary:

00000000: 00

11111111: 01

00000010: 10

00000011: 110

00000100: 111

©zyBooks 07/17/23 16:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

- 1) Compress the following: 00000000

00000000 11111111 00000100

**Check****Show answer**

- 2) Compress the following: 00000011

00000010

**Check****Show answer**

- 3) Decompress the following: 00 01 00

**Check****Show answer**

- 4) Does any code in the dictionary contain another code starting from the left of each code? Type yes or no. Ex: Consider a different dictionary having codes 1110 and 111; 1110 contains 111.



©zyBooks 07/17/23 16:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Check**Show answer**



- 5) Decompress the following, in which the spaces that were inserted above for reading convenience are absent:
0011000.

Check**Show answer**

©zyBooks 07/17/23 16:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Huffman coding

Huffman coding is a common compression technique that assigns fewer bits to frequent items, using a binary tree.

PARTICIPATION ACTIVITY

15.2.3: A binary tree can be used to determine the Huffman coding.



Animation captions:

1. Huffman coding first determines the frequencies of each item. Here, a occurs 4 times, b 3, c 2, and d 1. (Total is 10).
2. Each item is a "leaf node" in a tree. The pair of nodes yielding the lowest sum is found, and merged into a new node formed with that sum. Here, c and d yield $2 + 1 = 3$.
3. The merging continues. The lowest sum is b's 3 plus the new node's 3, yielding 6. (Note that c and d are no longer eligible nodes). The merging ends when only 1 node exists.
4. Each leaf node's encoding is obtained by traversing from the top node to the left. Each left branch appends a 0, and each right branch appends a 1, to the code.

When merging, if two (or more) different node pairs would yield the same sum, the choice among those pairs is arbitrary.

PARTICIPATION ACTIVITY

15.2.4: Huffman coding example: Frequency counts.



Given the text "seems he fled". Indicate the frequency counts.

©zyBooks 07/17/23 16:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



1) s

Check**Show answer**

2) e



Check**Show answer**

- 3) Each of m, h, f, l, and d

Check**Show answer**

- 4) (space)

Check**Show answer****PARTICIPATION ACTIVITY**

15.2.5: Huffman coding example: Merging nodes.



A 100-character text has these character frequencies:

- A: 50
- C: 40
- B: 4
- D: 3
- E: 3

- 1) What is the first merge?



- D and E: 6
- B and D: 7
- B and D and E: 10

- 2) What is the second merge?

©zyBooks 07/17/23 16:58 169246
Taylor Larrechea
COLORADOCSPB2270Summer2023

- B and D: 7
- DE and B: 10
- C and A: 90

- 3) What is the third merge?



- DEB and C: 40

DEB and C: 50

4) What is the fourth merge? □

None

DEBC and A: 100

5) What is the fifth merge? □

None

DEBCA and F

6) What is the code for A? □

0

1

7) What is the code for C? □

1

01

10

8) What is the code for B? □

001

110

9) What is the code for D? □

1110

1111

10) What is the code for E? □

1110

1111

11) 5 unique characters (A, B, C, D, E) can each be uniquely encoded in 3 bits (like 000, 001, 010, 011, and 100). With such a fixed-length code, how many bits are needed for the 100-character text?

100

©zyBooks 07/17/23 16:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

300

- 12) For the Huffman code determined in the above questions, the number of bits per character is A: 1, C: 2, B: 3, D: 4, and E: 4. Recalling the frequencies in the instructions, how many bits are needed for the 100-character text?

 14 166 300

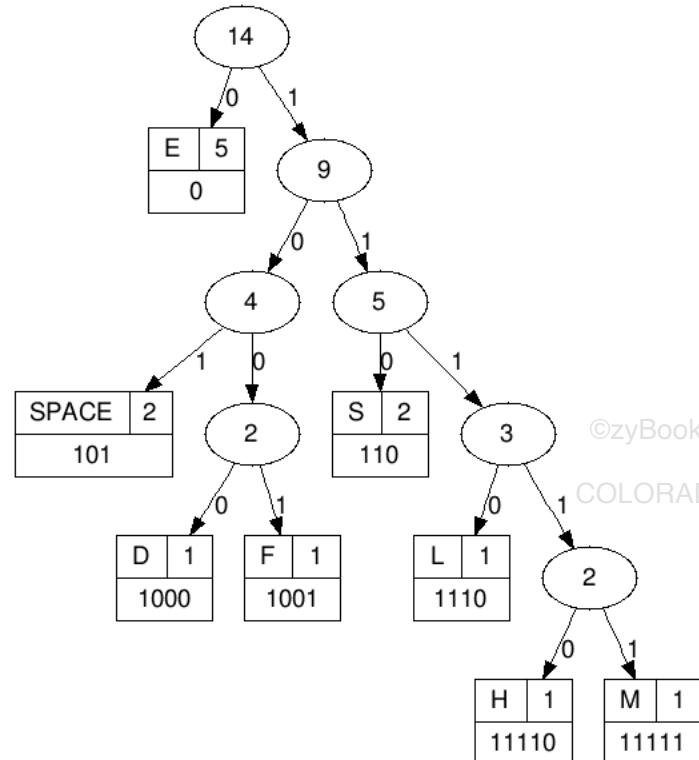
©zyBooks 07/17/23 16:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Note: For Huffman encoded data, the dictionary must be included along with the compressed data, to enable decompression. That dictionary adds to the total bits used. However, typically only large data files get compressed, so the dictionary is usually a tiny fraction of the total size.

Huffman tree web tools

[This site](#) has a tool that converts given text into a Huffman tree. For the earlier example of "seems he fled", the site generated the following tree.

Figure 15.2.1: Huffman tree for the text: SEEMS HE FLEED.



©zyBooks 07/17/23 16:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Source: [Huffman tree generator](#). No copyright held on generated images.

Table 15.2.1: Huffman and ASCII code table for the text: SEEMS HE FLEED.

©zyBooks 07/17/23 16:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Frequency	Chars	Huffman	ASCII
5	'E'	0	01000101
2	' '	101	00100000
2	'S'	110	01010011
1	'D'	1000	01000100
1	'F'	1001	01000110
1	'L'	1110	01001100
1	'H'	11110	01001000
1	'M'	11111	01001101

Note: [CrypTool Project](#) provides tools to generate a similar table comparing the sizes of Huffman and ASCII code.

For the text "SEEMS HE FLEED", Huffman code requires 39 bits while the ASCII code requires 112 bits.

- Huffman: 110 0 0 11111 110 101 11110 0 101 1001 1110 0 0 1000
- ASCII: 01010011 01000101 01000101 01001101 01010011 00100000 01001000 01000101
00100000 01000110 01001100 01000101 01000101 01000100

PARTICIPATION ACTIVITY

15.2.6: Huffman and ASCII code.

©zyBooks 07/17/23 16:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

1) E's Huffman code is ____ .

- 0
 01000101



2) ____ bits are needed to encode the 'SEEMS HE FLEED' using Huffman code.

- 39
- 112

©zyBooks 07/17/23 16:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Decompressing Huffman coded data

To decompress Huffman code data, one can use a Huffman tree and trace the branches for each bit, starting at the root. When the final node of the branch is reached, the result has been found. The process continues until the entire item is decompressed.

PARTICIPATION
ACTIVITY

15.2.7: Decompressing Huffman code.



Animation captions:

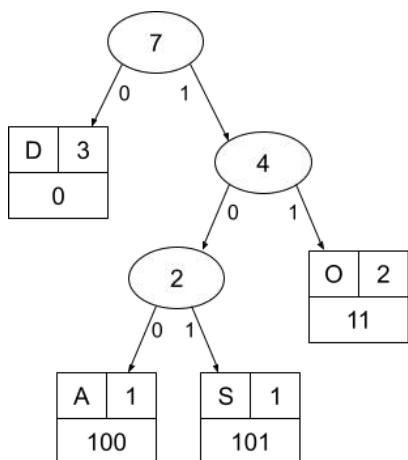
1. The Huffman code is decompressed by first starting at the root. The branches are followed for each bit.
2. When the final node of the branch is reached, the result has been found.
3. Once the final node is reached decoding restarts at the root node.
4. The process continues until the entire item is decompressed.

PARTICIPATION
ACTIVITY

15.2.8: Decompressing Huffman code.



Use the tree below to decompress 0111101000101.



©zyBooks 07/17/23 16:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

- 1) What is the first decoded character?



Check**Show answer**

- 2) What is the second decoded character?

 //

©zyBooks 07/17/23 16:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Check**Show answer**

- 3) 11 yields the third character O.
0 yields the fourth character D.

What is the next decoded character?

 //**Check****Show answer**

- 4) What is the decoded text?

 //**Check****Show answer**

Text files, images, and videos

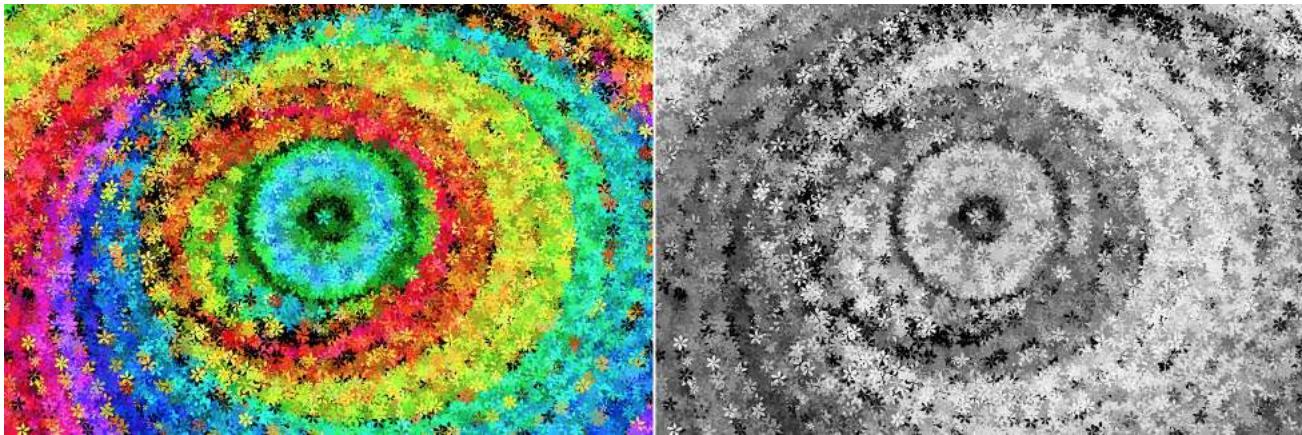
When compressing text files, an additional compression trick is used, wherein common sequences are treated as a pattern. Ex: "the box there is the right box" has the pattern "the" appear three times, and "box" twice. Thus, the dictionary may define a code for that pattern, so a code like 0010 might be listed not just as a letter like 't' but as a string like "the". Such a technique may be found in compression used to make ZIP files, for example.

Images may occupy much storage. Ex: An image with 1 million pixels and 3 bytes per pixel (for red, green, blue), may require 3 MB of storage. Each pixel is a number from 0 to 255. Some colors are much more common, like white (255, 255, 255) and black (0, 0, 0), so some numbers are much more common than others. Huffman coding is part of the common image compression technique known as JPEG. Image compression uses other techniques as well, which may lose some information (rounding, and discrete cosine transform, not discussed here) to achieve even greater compression.

Figure 15.2.2: Uncompressed image vs. a compressed image and color image vs. black and white image.



Source: zyBooks



Source: [Pixabay](#)

Video is a series of images known as frames. Thus, video compression techniques like MPEG include Huffman coding as well. Video compression also uses another compression technique: Because successive frames have only small differences, after an image, several successive frames may be represented just by the differences from the previous frame.

PARTICIPATION ACTIVITY

15.2.9: Text files, image, and video compression. ©zyBooks 07/17/23 16:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 1) When compressing text, a dictionary may produce a code for a single character or a string.

- True
 False



- 2) A compressed image takes up the same amount of storage as an uncompressed image.
- True
 - False
- 3) A common video compression technique involves only changing components that are different from the previous video frame.

- True
- False

©zyBooks 07/17/23 16:58 169246
Taylor Larrechea
COLORADOCSPB2270Summer2023

Exploring further:

- [Huffman coding](#) (Wikipedia)
- [Huffman tree generator](#) (huffman.ooz.ie)
- [Comparison of Huffman code and ASCII code](#) (cyrptool-online.org)

©zyBooks 07/17/23 16:58 169246
Taylor Larrechea
COLORADOCSPB2270Summer2023

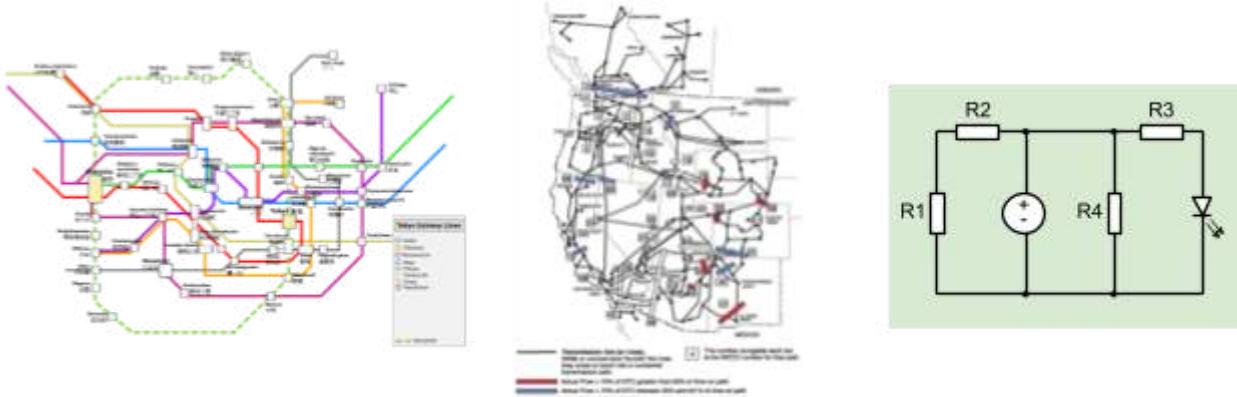
16.1 Graphs: Introduction

Introduction to graphs

Many items in the world are connected, such as computers on a network connected by wires, cities connected by roads, or people connected by friendship.

©zyBooks 07/19/23 22:03 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Figure 16.1.1: Examples of connected items: Subway map, electrical power transmission, electrical circuit.



Source: Subway map ([Comicinker \(Own work\)](#) / [CC-BY-SA-3.0](#) via Wikimedia Commons), Internet map ([Department of Energy](#) / Public domain via Wikimedia Commons), Electrical circuit (zyBooks)

A **graph** is a data structure for representing connections among items, and consists of vertices connected by edges.

- A **vertex** (or node) represents an item in a graph.
- An **edge** represents a connection between two vertices in a graph.

PARTICIPATION ACTIVITY

16.1.1: A graph represents connections among items, like among computers, or people.

©zyBooks 07/19/23 22:03 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Animation captions:

1. Items in the world may have connections, like a computer network.
2. A graph's vertices represent items.
3. A graph's edges represent connections.

4. A graph can represent many different things, like friendships among people. Raj and Maya are friends, but Raj and Jen are not.

For a given graph, the number of vertices is commonly represented as V, and the number of edges as E.

PARTICIPATION ACTIVITY**16.1.2: Graph basics.**

©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



Refer to the above graphs.

- 1) The computer network graph has how many vertices?

- 5
- 6

- 2) The computer network graph has how many edges?

- 5
- 6

- 3) Are Maya and Thuy friends?

- Yes
- No

- 4) Can a vertex have more than one edge?

- Yes
- No

- 5) Can an edge connect more than two vertices?

- Yes
- No

- 6) Given 4 vertices A, B, C, and D and at most one edge between a vertex pair, what is the maximum number of edges?

- 6
- 16

©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



Adjacency and paths

In a graph:

- Two vertices are **adjacent** if connected by an edge.
- A **path** is a sequence of edges leading from a source (starting) vertex to a destination (ending) vertex. The **path length** is the number of edges in the path.
- The **distance** between two vertices is the number of edges on the shortest path between those vertices.

©zyBooks 07/19/23 22:03 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

16.1.3: Graphs: adjacency, paths, and distance.



Animation captions:

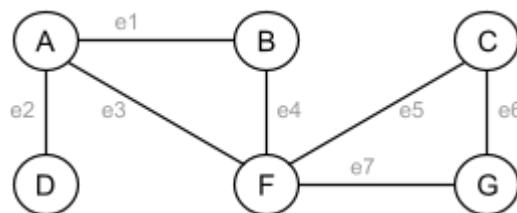
1. Two vertices are adjacent if connected by an edge. PC1 and Server1 are adjacent. PC1 and Server3 are not adjacent.
2. A path is a sequence of edges from a source vertex to a destination vertex.
3. A path's length is the path's number of edges. Vertex distance is the length of the shortest path: Distance from PC1 to PC2 is 2.

PARTICIPATION ACTIVITY

16.1.4: Graph properties.



Refer to the following graph.



- 1) A and B are adjacent.

- True
 False



- 2) A and C are adjacent.

- True
 False

©zyBooks 07/19/23 22:03 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



- 3) Which of the following is a path from B to G?

- e3, e7



- e1, e3, e7
 - No path from B to G.
- 4) What is the distance from D to C?
- 3
 - 4
 - 5

©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

16.2 Applications of graphs

Geographic maps and navigation

Graphs are often used to represent a geographic map, which can contain information about places and travel routes. Ex: Vertices in a graph can represent airports, with edges representing available flights. Edge weights in such graphs often represent the length of a travel route, either in total distance or expected time taken to navigate the route. Ex: A map service with access to real-time traffic information can assign travel times to road segments.

PARTICIPATION
ACTIVITY

16.2.1: Driving directions use graphs and shortest path algorithms to determine the best route from start to destination.



Animation content:

undefined

Animation captions:

1. A road map can be represented by a graph. Each intersection of roads is a vertex. Destinations like the beach or a house are also vertices.
2. Roads between vertices are edges. A map service with realtime traffic information can assign travel times as edge weights.
3. A shortest path finding algorithm can be used to find the shortest path starting at the house and ending at the beach.

©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

16.2.2: Using graphs for road navigation.



- 1) The longer a street is, the more vertices will be needed to represent that street.

True
 False

- 2) When using the physical distance between vertices as edge weights, a shortest path algorithm finds the fastest route of travel.

True
 False

- 3) Navigation software would have no need to place a vertex on a road in a location where the road does not intersect any other roads.

True
 False

- 4) If navigation software uses GPS to automatically determine the start location for a route, the vertex closest to the GPS coordinates can be used as the starting vertex.

True
 False



©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

**PARTICIPATION ACTIVITY**

16.2.3: Using graphs for flight navigation.



Suppose a graph is used to represent airline flights. Vertices represent airports and edge weights represent flight durations.

©zyBooks 07/19/23 22:03 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 1) The weight of an edge connecting two airport vertices may change based on

_____.
 flight delays
 weather conditions



flight cost

- 2) Edges in the graph could potentially be added or removed during a single day's worth of flights.

True

False

©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



Product recommendations

A graph can be used to represent relationships between products. Vertices in the graph corresponding to a customer's purchased products have adjacent vertices representing products that can be recommended to the customer.

PARTICIPATION ACTIVITY

16.2.4: A graph of product relationships can be used to produce recommendations based on purchase history.



Animation content:

undefined

Animation captions:

1. An online shopping service can represent relationships between products being sold using a graph.
2. Relationships may be based on the products alone. A game console requires a TV and games, so the game console vertex connects to TV and game products.
3. Vertices representing common kitchen products, such as a blender, dishwashing soap, kitchen towels, and oven mitts, are also connected.
4. Connections might also represent common purchases that occur by chance. Maybe several customers who purchase a Blu-ray player also purchase a blender.
5. A customer's purchases can be linked to the product vertices in the graph.
6. Adjacent vertices can then be used to provide a list of recommendations to the customer.

©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



PARTICIPATION ACTIVITY

16.2.5: Product recommendations.

- 1) If a customer buys only a Blu-ray player, which product is not likely to be recommended?

Television 1 or 2



- Blender
- Game console
- 2) Which single purchase would produce the largest number of recommendations for the customer?
- Tablet computer
- Blender
- Game console
- 3) If "secondary recommendations" included all products adjacent to recommended products, which products would be secondary recommendations after buying oven mitts?
- Blender, kitchen towels, and muffin pan
- Dishwashing soap, Blu-ray player, and muffin mix
- Game console and tablet computer case

©zyBooks 07/19/23 22:03 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Social and professional networks

A graph may use a vertex to represent a person. An edge in such a graph represents a relationship between 2 people. In a graph representing a social network, an edge commonly represents friendship. In a graph representing a professional network, an edge commonly represents business conducted between 2 people.

PARTICIPATION ACTIVITY

16.2.6: Professional networks represented by graphs help people establish business connections.

©zyBooks 07/19/23 22:03 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Animation content:

undefined

Animation captions:

1. In a graph representing a professional network, vertices represent people and edges represent a business connection.
2. Tuyet conducts business with Wilford, adding a new edge in the graph. Similarly, Manuel conducts business with Keira.
3. The graph can help professionals find new connections. Shayla connects with Octavio through a mutual contact, Manuel.

©zyBooks 07/19/23 22:03 1692462

PARTICIPATION ACTIVITY**16.2.7: Professional network.**Taylor Larrechea
COLORADOCSPB2270Summer2023

Refer to the animation's graph representing the professional network.

- 1) Who has conducted business with Eusebio?
 - Giovanna
 - Manuel
 - Keira
- 2) If Octavio wishes to connect with Wilford, who is the best person to introduce the 2 to each other?
 - Shayla
 - Manuel
 - Rita
- 3) What is the length of the shortest path between Salvatore and Reva?
 - 5
 - 6
 - 7

16.3 Graph representations: Adjacency lists

©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea
COLORADOCSPB2270Summer2023

Adjacency lists

Various approaches exist for representing a graph data structure. A common approach is an adjacency list. Recall that two vertices are **adjacent** if connected by an edge. In an **adjacency list** graph representation, each vertex has a list of adjacent vertices, each list item representing an edge.

**Animation captions:**

1. Each vertex has a list of adjacent vertices for edges. The edge connecting A and B appears in A's list and also in B's list.
2. Each edge appears in the lists of the edge's two vertices.

©zyBooks 07/19/23 22:03 1692462

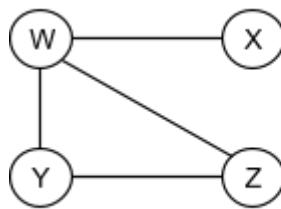
Taylor Larrechea

COLORADOCSPB2270Summer2023

Advantages of adjacency lists

A key advantage of an adjacency list graph representation is a size of $O(V + E)$, because each vertex appears once, and each edge appears twice. V refers to the number of vertices, E the number of edges.

However, a disadvantage is that determining whether two vertices are adjacent is $O(V)$, because one vertex's adjacency list must be traversed looking for the other vertex, and that list could have V items. However, in most applications, a vertex is only adjacent to a small fraction of the other vertices, yielding a sparse graph. A **sparse graph** has far fewer edges than the maximum possible. Many graphs are sparse, like those representing a computer network, flights between cities, or friendships among people (every person isn't friends with every other person). Thus, the adjacency list graph representation is very common.



Vertices	Adjacent vertices
W	?
X	W
Y	?
Z	?

- 1) What are vertex W's adjacent vertices?



- Y, Z
- X, Y, Z

©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

- 2) What are vertex Y's adjacent vertices?



- W, X, Z
- W, Z
- Z



3) What are vertex Z's adjacent vertices?

- W
- W, X
- W, Y

PARTICIPATION ACTIVITY

16.3.3: Adjacency lists: Bus routes.

©zyBooks 07/19/23 22:03 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



The following adjacency list represents bus routes. Ex: A commuter can take the bus from Belmont to Sonoma.

Vertices	Adjacent vertices (edges)		
Belmont	Marin City	Sonoma	
Hillsborough	Livermore	Marin City	Sonoma
Livermore	Hillsborough	Sonoma	
Marin City	Belmont	Hillsborough	Sonoma
Sonoma	Belmont	Hillsborough	Livermore
			Marin City

1) A direct bus route exists from Belmont to Sonoma or Belmont to

//
Check**Show answer**

2) How many buses are needed to go from Livermore to Hillsborough?

//
Check**Show answer**

3) What is the minimum number of buses needed to go from Belmont to Hillsborough?

//
Check**Show answer**

©zyBooks 07/19/23 22:03 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023





4) Is the following a path from Livermore to Marin City? Type: Yes or No
Livermore, Hillsborough, Sonoma, Marin City

Check

Show answer

©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

CHALLENGE ACTIVITY

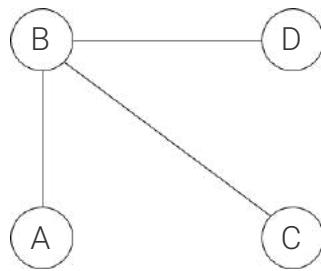
16.3.1: Adjacency lists.



489394.3384924.qx3zqy7

Start

Select the missing vertices to complete the adjacency list representation of the given graph.



Vertices	Adjacent vertices (edges)
A	(1)
B	A C (2)
C	(3)
D	(4)

- (1): ✓
- (2): ✓
- (3): ✓
- (4): ✓

1

2

Check

Next

©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

16.4 Graph representations: Adjacency matrices

Adjacency matrices

Various approaches exist for representing a graph data structure. One approach is an adjacency matrix. Recall that two vertices are **adjacent** if connected by an edge. In an **adjacency matrix** graph representation, each vertex is assigned to a matrix row and column, and a matrix element is 1 if the corresponding two vertices have an edge or is 0 otherwise.

PARTICIPATION ACTIVITY

16.4.1: Adjacency matrix representation.


©zyBooks 07/19/23 22:03 1692462
Taylor Larrechea
COLORADOCSBP2270Summer2023
Animation captions:

1. Each vertex is assigned to a row and column.
2. An edge connecting A and B is a 1 in A's row and B's column.
3. Similarly, that same edge is a 1 in B's row and A's column. (The matrix will thus be symmetric.)
4. Each edge similarly has two 1's in the matrix. Other matrix elements are 0's (not shown).

Analysis of adjacency matrices

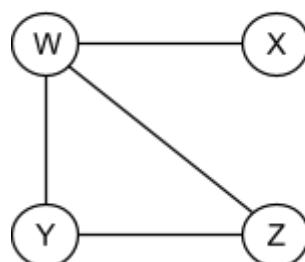
Assuming the common implementation as a two-dimensional array whose elements are accessible in $O(1)$, then an adjacency matrix's key benefit is $O(1)$ determination of whether two vertices are adjacent: The corresponding element is just checked for 0 or 1.

A key drawback is $O(V^2)$ size. Ex: A graph with 1000 vertices would require a 1000×1000 matrix, meaning 1,000,000 elements. An adjacency matrix's large size is inefficient for a sparse graph, in which most elements would be 0's.

An adjacency matrix only represents edges among vertices; if each vertex has data, like a person's name and address, then a separate list of vertices is needed.

PARTICIPATION ACTIVITY

16.4.2: Adjacency matrix.



	W	X	Y	Z
W	0	(a)	(b)	1
X	1	0	0	(c)
Y	(d)	0	0	(e)
Z	1	0	(f)	0

©zyBooks 07/19/23 22:03 1692462
 Taylor Larrechea
 COLORADOCSBP2270Summer2023

1) (a)


 0

 1

2) (b)



0 1

3) (c)

 0 1

©zyBooks 07/19/23 22:03 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

4) (d)

 0 1

5) (e)

 0 1

6) (f)

 0 1**PARTICIPATION ACTIVITY**

16.4.3: Adjacency matrix: Power grid map.



The following adjacency matrix represents the map of a city's electrical power grid. Ex:
Sector A's power grid is connected to Sector B's power grid.

	A	B	C	D	E
A	0	1	1	1	0
B	1	0	1	1	1
C	1	1	0	1	1
D	1	1	1	0	0
E	0	1	1	0	0

©zyBooks 07/19/23 22:03 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

1) How many edges does D have?


Check**Show answer**



- 2) How many edges does the graph contain?

Check
[Show answer](#)

- 3) Assume Sector D has a power failure. Can power from Sector A be diverted directly to Sector D? Type: Yes or No

Check
[Show answer](#)

©zyBooks 07/19/23 22:03 169246
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 4) Assume Sector E has a power failure. Can power from Sector A be diverted directly to Sector E? Type: Yes or No

Check
[Show answer](#)

- 5) Is the following a path from Sector A to E? Type: Yes or No
AD, DB, BE

Check
[Show answer](#)

CHALLENGE ACTIVITY

16.4.1: Adjacency matrices.



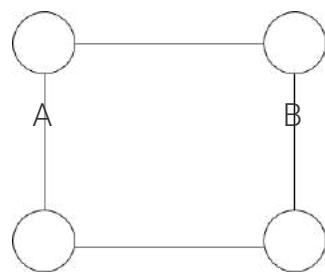
489394.3384924.qx3zqy7

Start

©zyBooks 07/19/23 22:03 169246
Taylor Larrechea
COLORADOCSPB2270Summer2023

Select the missing elements to complete the adjacency matrix representation of the given graph.

		A	B	C	D
D		A	0	1	(w)
	C	B	1	0	(x)
					0



zyBooks

C	0	1	0	(y)
D	1	0	(z)	0

- (w):
- (x):
- (y):
- (z):

©zyBooks 07/19/23 22:03 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

1

2

Check**Next**

16.5 Graphs: Breadth-first search

Graph traversal and breadth-first search

An algorithm commonly must visit every vertex in a graph in some order, known as a **graph traversal**.

A **breadth-first search** (BFS) is a traversal that visits a starting vertex, then all vertices of distance 1 from that vertex, then of distance 2, and so on, without revisiting a vertex.

PARTICIPATION ACTIVITY

16.5.1: Breadth-first search.



Animation captions:

1. Breadth-first search starting at A visits vertices based on distance from A. B and D are distance 1.
2. E and F are distance 2 from A. Note: A path of length 3 also exists to F, but distance metric uses shortest path.
3. C is distance 3 from A.
4. Breadth-first search from A visits A, then vertices of distance 1, then 2, then 3. Note: Visiting order of same-distance vertices doesn't matter.

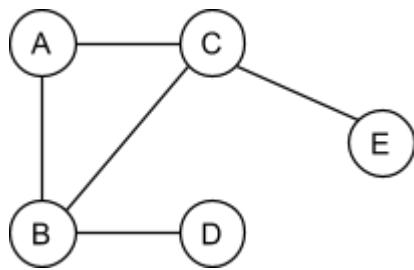
©zyBooks 07/19/23 22:03 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

16.5.2: Breadth-first search traversal.



Perform a breadth-first search of the graph below. Assume the starting vertex is E.



- 1) Which vertex is visited first?

 //

©zyBooks 07/19/23 22:03 169246
Taylor Larrechea
COLORADOCSPB2270Summer2023

Check**Show answer**

- 2) Which vertex is visited second?

 //**Check****Show answer**

- 3) Which vertex is visited third?

 //**Check****Show answer**

- 4) What is C's distance?

 //**Check****Show answer**

- 5) What is D's distance?

 //**Check****Show answer**

- 6) The BFS traversal of a graph is unique. Type: Yes or No

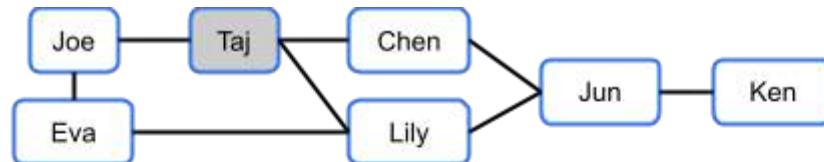
©zyBooks 07/19/23 22:03 169246
Taylor Larrechea
COLORADOCSPB2270Summer2023

 //**Check****Show answer**

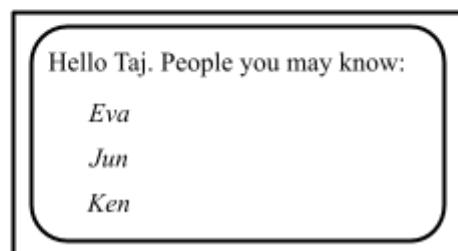
Example: Social networking connection recommender

Example 16.5.1: Social networking connection recommender using breadth-first search.

Social networking sites like Facebook and LinkedIn use graphs to represent connections among people. For a particular user, a site may wish to recommend new connections. One approach does a breadth-first search starting from the user, recommending new connections starting at distance 2 (distance 1 people are already connected with the user).



Breadth-first traversal:	Taj	Joe	Chen	Lily	Eva	Jun	Ken
	0	1	1	1	2	2	3



PARTICIPATION ACTIVITY

16.5.3: BFS: Connection recommendation.



Refer to the connection recommendation example above.

- 1) A distance greater than 0 indicates people are not connected with the user.



- True
- False

- 2) People with a distance of 2 are recommended before people with a distance of 3.



- True
- False

©zyBooks 07/19/23 22:03 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



- 3) If Chen is the user, the system also recommends Eva, Jun, and Ken.

- True
- False

Example: Find closest item in a peer-to-peer network

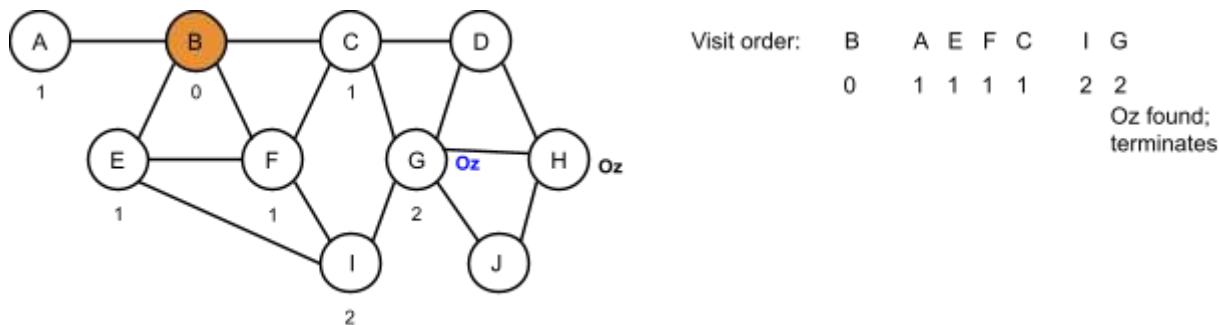
©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Example 16.5.2: Application of BFS: Find closest item in a peer-to-peer network.

In a **peer-to-peer network**, computers are connected via a network and may seek and download file copies (such as songs or movies) via intermediary computers or routers. For example, one computer may seek the movie "The Wizard of Oz", which may exist on 10 computers in a network consisting of 100,000 computers. Finding the closest computer (having the fewest intermediaries) yields a faster download. A BFS traversal of the network graph can find the closest computer with that movie. The BFS traversal can be set to immediately return if the item sought is found during a vertex visit. Below, visiting vertex G finds the movie; BFS terminates, and a download process can begin, involving a path length of 2 (so only 1 intermediary). Vertex H also has the movie, but is further from B so wasn't visited yet during BFS. (Note: Distances of vertices visited during the BFS from B are shown below for convenience).



PARTICIPATION ACTIVITY

16.5.4: BFS application: Peer-to-peer search.

©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



Consider the above peer-to-peer example.

- 1) In some BFS traversals starting from vertex B, vertex H may be visited before vertex G.
 - True

False

- 2) If vertex J sought the movie Oz, the download might occur from either G or H.

 True False

- 3) If vertex E sought the movie Oz, the download might occur from either G or H.

 True False

©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Breadth-first search algorithm

An algorithm for breadth-first search enqueues the starting vertex in a queue. While the queue is not empty, the algorithm dequeues a vertex from the queue, visits the dequeued vertex, enqueues that vertex's adjacent vertices (if not already discovered), and repeats.

PARTICIPATION ACTIVITY

16.5.5: BFS algorithm.

Animation content:

undefined

Animation captions:

1. BFS enqueues the start vertex (in this case A) in frontierQueue, and adds A to discoveredSet.
2. Vertex A is dequeued from frontierQueue and visited.
3. Undiscovered vertices adjacent to A are enqueueed in frontierQueue and added to discoveredSet.
4. Vertex A's visit is complete. The process continues on to next vertices in the frontierQueue, D and B.
5. E is visited. Vertices B and F are adjacent to E, but are already in discoveredSet. So B and F are not again added to frontierQueue or discoveredSet.
6. The process continues until frontierQueue is empty. discoveredSet shows the visit order. Note that each vertex's distance from start is shown on the graph.

©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

When the BFS algorithm first encounters a vertex, that vertex is said to have been **discovered**. In the BFS algorithm, the vertices in the queue are called the **frontier**, being vertices thus far discovered but

not yet visited. Because each vertex is visited at most once, an already-discovered vertex is not enqueued again.

A "visit" may mean to print the vertex, append the vertex to a list, compare vertex data to a value and return the vertex if found, etc.

PARTICIPATION ACTIVITY

16.5.6: BFS algorithm.

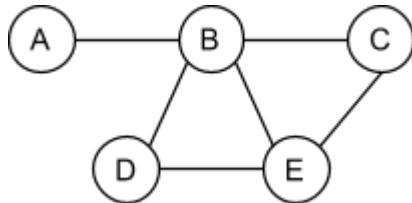


©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

BFS is run on the following graph. Assume C is the starting vertex.



- 1) Which vertices are in the frontierQueue before the first iteration of the while loop?

- A
- C
- A, B, C, D, E

- 2) Which vertices are in discoveredSet after the first iteration of the while loop?

- C, B, E
- B, E
- C

- 3) In the second iteration, currentV = B. Which vertices are in discoveredSet after the second iteration of the while loop?

- C, B, E, A
- B, E, A, D
- C, B, E, A, D

- 4) In the second iteration, currentV = B. Which vertices are in frontierQueue after the second iteration of the while loop?



©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

- C, B, E, A, D
 - E, A, D
 - frontierQueue is empty
- 5) BFS terminates after the second iteration, because all vertices are in the discoveredSet.
- True
 - False

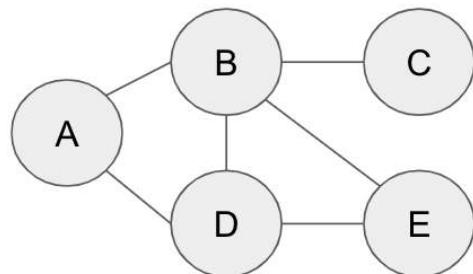
©zyBooks 07/19/23 22:03 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

CHALLENGE ACTIVITY**16.5.1: Breadth-first search.**

489394.3384924.qx3zqy7

Start

Given the graph below and the starting vertex B:



What is D's distance? Ex: 5

What is B's distance?

What is A's distance?

What is E's distance?

What is C's distance?

©zyBooks 07/19/23 22:03 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

1

2

3

4

[Check](#)[Next](#)

16.6 Graphs: Depth-first search

©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Graph traversal and depth-first search

An algorithm commonly must visit every vertex in a graph in some order, known as a **graph traversal**. A **depth-first search** (DFS) is a traversal that visits a starting vertex, then visits every vertex along each path starting from that vertex to the path's end before backtracking.

PARTICIPATION ACTIVITY

16.6.1: Depth-first search.



Animation captions:

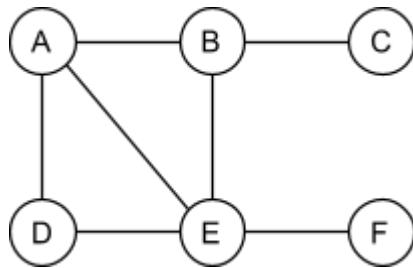
1. Depth-first search starting at A descends along a path to the path's end before backtracking.
2. Reach path's end: Backtrack to F, visit F's other adjacent vertex (E). B already visited, backtrack again.
3. Backtracked all the way to A. Visit A's other adjacent vertex (D). No other adjacent vertices: Done.

PARTICIPATION ACTIVITY

16.6.2: Depth-first search traversal.



Perform a depth-first search of the graph below. Assume the starting vertex is E.

©zyBooks 07/19/23 22:03 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 1) Which vertex is visited first?

[Check](#)[Show answer](#)



- 2) Assume DFS traverses the following vertices: E, A, B. Which vertex is visited next?

 //**Check****Show answer**

- 3) Assume DFS traverses the following vertices: E, A, B, C. Which vertex is visited next?

 //**Check****Show answer**

- 4) Is the following a valid DFS traversal? Type: Yes or No
E, D, F, A, B, C

 //**Check****Show answer**

- 5) Is the following a valid DFS traversal? Type: Yes or No
E, D, A, B, C, F

 //**Check****Show answer**

- 6) The DFS traversal of a graph is unique. Type: Yes or No

 //**Check****Show answer**

©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



Depth-first search algorithm

An algorithm for depth-first search pushes the starting vertex to a stack. While the stack is not empty, the algorithm pops the vertex from the top of the stack. If the vertex has not already been visited, the algorithm visits the vertex and pushes the adjacent vertices to the stack.

PARTICIPATION ACTIVITY

16.6.3: Depth-first search traversal with starting vertex A.

**Animation content:**

undefined

©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

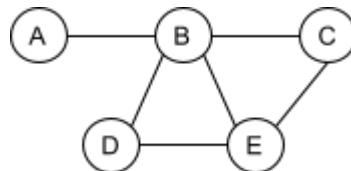
1. A depth-first search at vertex A first pushes vertex A onto the stack. The first loop iteration then pops vertex A off the stack and assigns currentV with that vertex.
2. Vertex A is visited and added to the visited set. Each vertex adjacent to A is pushed onto the stack.
3. Vertex B is popped off the stack and processed as the next currentV.
4. Vertices F and E are processed similarly.
5. Vertices F and B are in the visited set and are skipped after being popped off the stack.
6. Vertex C is popped off the stack and visited. All remaining vertices except D are in the visited set.
7. Vertex D is the last vertex visited.

PARTICIPATION ACTIVITY

16.6.4: DFS algorithm.



DFS is run on the following graph. Assume C is the starting vertex.



- 1) Which vertices are in the stack before the first iteration of the while loop?



- A
- C
- A, B, C, D, E

- 2) Which vertices are in the stack after the first iteration of the while loop?



- B, E, C
- B, E
- B

©zyBooks 07/19/23 22:03 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



- 3) Which vertices are in visitedSet after the first iteration of the while loop?

- B, E, C
- C
- B

- 4) In the second iteration, currentV = B.
Which vertices are in the stack after the second iteration of the while loop?

- C
- A, C, D, E
- A, C, D, E, E

- 5) Which vertices are in visitedSet after the second iteration of the while loop?

- C, B
- C
- B

- 6) DFS terminates once all vertices are added to visitedSet.

- True
- False

©zyBooks 07/19/23 22:03 169246
Taylor Larrechea
COLORADOCSPB2270Summer2023

Recursive DFS algorithm

A recursive DFS can be implemented using the program stack instead of an explicit stack. The recursive DFS algorithm is first called with the starting vertex. If the vertex has not already been visited, the recursive algorithm visits the vertex and performs a recursive DFS call for each adjacent vertex.

Figure 16.6.1: Recursive depth-first search.

©zyBooks 07/19/23 22:03 169246
Taylor Larrechea
COLORADOCSPB2270Summer2023

```
RecursiveDFS(currentV) {  
    if ( currentV is not in visitedSet ) {  
        Add currentV to visitedSet  
        "Visit" currentV  
        for each vertex adjV adjacent to  
        currentV {  
            RecursiveDFS(adjV)  
        }  
    }  
}
```

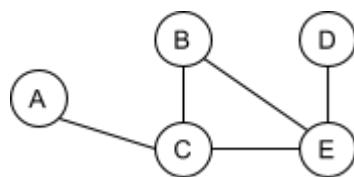
©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY**16.6.5: Recursive DFS algorithm.**

The recursive DFS algorithm is run on the following graph. Assume D is the starting vertex.



- 1) The recursive DFS algorithm uses a queue to determine which vertices to visit.

- True
- False

- 2) DFS begins with the function call RecursiveDFS(D).

- True
- False

- 3) If B is not yet visited, RecursiveDFS(B) will make subsequent calls to RecursiveDFS(C) and RecursiveDFS(E).

- True
- False

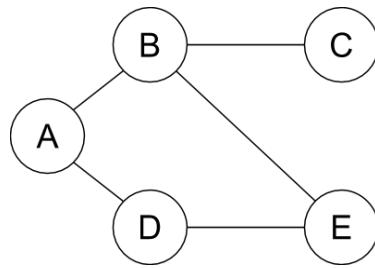
©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

CHALLENGE ACTIVITY**16.6.1: Graphs: Depth-first search.**

489394.3384924.qx3zqy7

[Start](#)

©zyBooks 07/19/23 22:03 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Enter a valid depth-first search traversal when C is the starting vertex.

Ex: A, B, C, D, E (commas between values)

1

2

3

4

5

[Check](#)

[Next](#)

16.7 Directed graphs

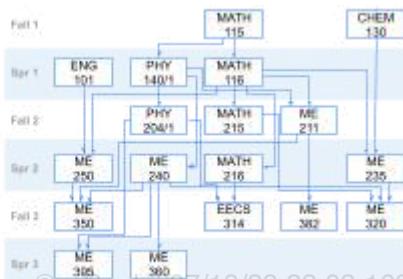
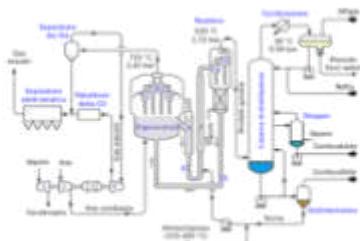
Directed graphs

A **directed graph**, or **digraph**, consists of vertices connected by directed edges. A **directed edge** is a connection between a starting vertex and a terminating vertex. In a directed graph, a vertex Y is **adjacent** to a vertex X, if there is an edge from X to Y.

Many graphs are directed, like those representing links between web pages, maps for navigation, or college course prerequisites.

©zyBooks 07/19/23 22:03 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Figure 16.7.1: Directed graph examples: Process flow diagram, airline routes, and college course prerequisites.



©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Geez-oZ

Source: Fluid catalytic cracker ([Mbeychok](#) / Public Domain via Wikimedia Commons), Florida airline routes 1974 ([Geez-oZ](#) (Own work) / [CC-BY-SA-3.0](#) via Wikimedia Commons), college course prerequisites (zyBooks)

PARTICIPATION ACTIVITY

16.7.1: A directed graph represents connections among items, like links between web pages, or airline routes.



Animation captions:

1. Items in the world may have directed connections, like links on a website.
2. A directed graph's vertices represent items.
3. Vertices are connected by directed edges.
4. A directed edge represents a connection from a starting vertex to a terminating vertex; the terminating vertex is adjacent to the starting vertex. B is adjacent to A, but A is not adjacent to B.
5. A directed graph can represent many things, like airline connections between cities. A flight is available from Los Angeles to Tucson, but not Tucson to Los Angeles.

PARTICIPATION ACTIVITY

16.7.2: Directed graph basics.



Refer to the above graphs.

- 1) E is a ____ in the directed graph.



- vertex
- directed edge

- 2) A directed edge connects vertices A and D. D is the ____ vertex.



- starting
- terminating

- 3) The airline routes graph is a digraph.



©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

- True
- False

4) Tucson is adjacent to ____.

- San Francisco
- Los Angeles
- Dallas

©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Paths and cycles

In a directed graph:

- A **path** is a sequence of directed edges leading from a source (starting) vertex to a destination (ending) vertex.
- A **cycle** is a path that starts and ends at the same vertex. A directed graph is **cyclic** if the graph contains a cycle, and **acyclic** if the graph does not contain a cycle.

PARTICIPATION
ACTIVITY

16.7.3: Directed graph: Paths and cycles.

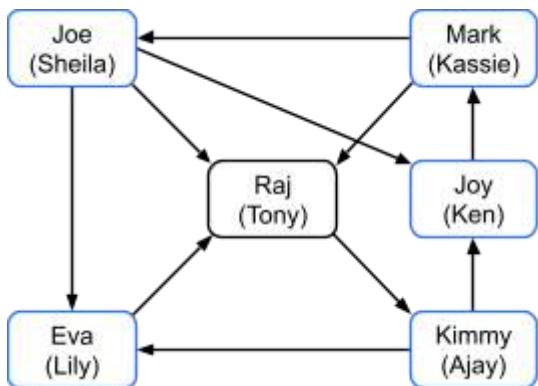
Animation captions:

1. A path is a sequence of directed edges leading from a source (starting) vertex to a destination (ending) vertex.
2. A cycle is a path that starts and ends at the same vertex. A graph can have more than one cycle.
3. A cyclic graph contains a cycle. An acyclic graph contains no cycles.

Example 16.7.1: Cycles in directed graphs: Kidney transplants.

A patient needing a kidney transplant may have a family member willing to donate a kidney but is incompatible. That family member is willing to donate a kidney to someone else, as long as their family member also receives a kidney donation. Suppose Gregory needs a kidney. Gregory's wife, Eleanor, is willing to donate a kidney but is not compatible with Gregory. However, Eleanor is compatible with another patient Joanna, and Joanna's husband Darrell is compatible with Gregory. So, Eleanor donates a kidney to Joanna, and Darrell donates a kidney to Gregory, which is an example of a 2-way kidney transplant. In 2015, a 9-way kidney transplant involving 18 patients was performed within 36 hours (Source: SF

Gate). Multiple-patient kidney transplants can be represented as cycles within a directed graph.



3-way transplant: Eva (Lily), Raj (Tony), Kimmy (Ajay)

4-way transplant: Raj (Tony), Kimmy (Ajay), Joy (Ken), Mark (Kassie)

©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADO CSPB2270Summer2023

In this graph, vertices represent patients, and edges represent compatibility between a patient's family member (shown in parentheses) and another patient. An N-way kidney transplant is represented as a cycle with N edges. Due to the complexity of coordinating multiple simultaneous surgeries, hospitals and doctors typically try to find the shortest possible cycle.

PARTICIPATION ACTIVITY

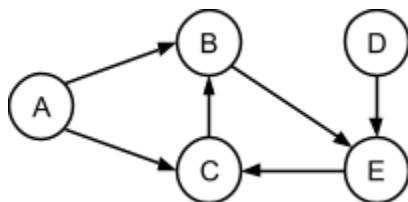
16.7.4: Directed graphs: Cyclic and acyclic.



Determine if each of the following graphs is cyclic or acyclic.



1)

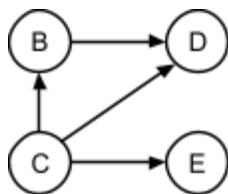


Cyclic

Acyclic



2)

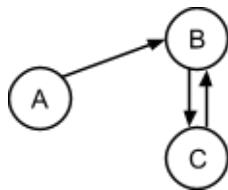


Cyclic

Acyclic



3)



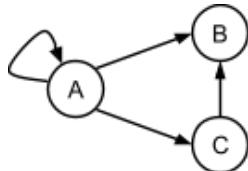
©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADO CSPB2270Summer2023

- Cyclic
- Acyclic

4)



- Cyclic
- Acyclic



©zyBooks 07/19/23 22:03 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

CHALLENGE ACTIVITY

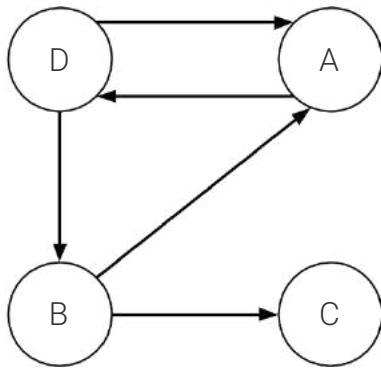
16.7.1: Directed graphs.



489394.3384924.qx3zqy7

Start

Consider the following directed graph.



Identify the vertices (if any) that are adjacent to:

B

Ex: A, B

Enter values separated by commas.

If none, enter: none

C

D

1

2

©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Check**Next**

16.8 Weighted graphs

Weighted graphs

A **weighted graph** associates a weight with each edge. A graph edge's **weight**, or **cost**, represents some numerical value between vertex items, such as flight cost between airports, connection speed between computers, or travel time between cities. A weighted graph may be directed or undirected.

PARTICIPATION ACTIVITY

16.8.1: Weighted graphs associate weight with each edge.

©zyBooks 07/19/23 22:03 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



Animation captions:

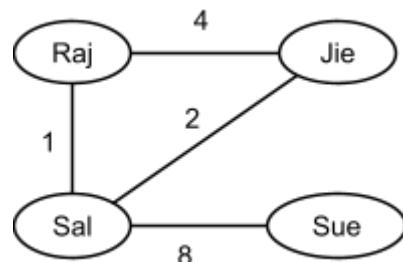
1. A weighted graph associates a numerical weight, or cost, with each edge. Ex: Edge weights may indicate connection speed (Mbps) between computers.
2. Weighted graphs can be directed. Ex: Edge weights may indicate travel time (hours) between cities; travel times may vary by direction.

PARTICIPATION ACTIVITY

16.8.2: Weighted graphs.



- 1) This graph's weights indicate the number of times coworkers have collaborated on projects. How many times have Raj and Jie teamed up?

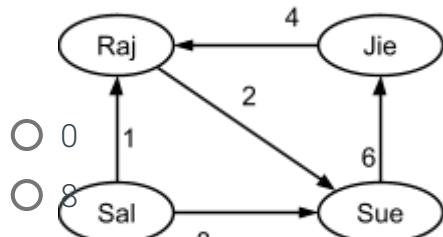


- 1
- 4

- 2) This graph indicates the number of times a worker has nominated a coworker for an award. How many times has Sal nominated Sue?

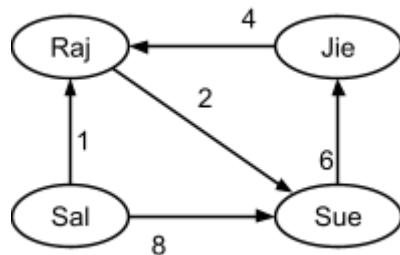
©zyBooks 07/19/23 22:03 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023





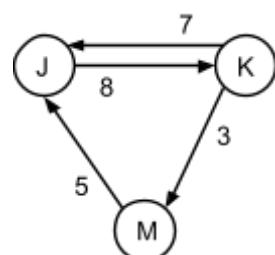
©zyBooks 07/19/23 22:03 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

- 3) This graph indicates the number of times a worker has nominated a coworker for an award. How many times has Raj nominated Jie?



- 4
- 1
- 0

- 4) The weight of the edge from K to J is _____.



©zyBooks 07/19/23 22:03 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

- 7
- 8

Path length in weighted graphs

In a weighted graph, the **path length** is the sum of the edge weights in the path.

PARTICIPATION ACTIVITY

16.8.3: Path length is the sum of edge weights.

**Animation captions:**

1. A path is a sequence of edges from a source vertex to a destination vertex.
2. The path length is the sum of the edge weights in the path.
3. The shortest path is the path yielding the lowest sum of edge weights. Ex: The shortest path from Paris to Marseille is 6.

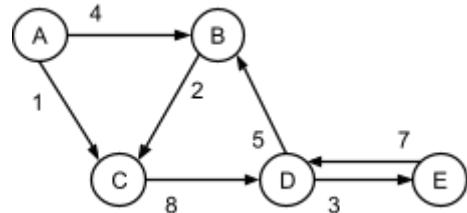
©zyBooks 07/19/23 22:03 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

16.8.4: Path length and shortest path.

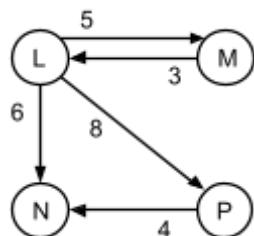


- 1) Given a path A, C, D, the path length is ____.



//
Check**Show answer**

- 2) The shortest path from M to N has a length of ____.

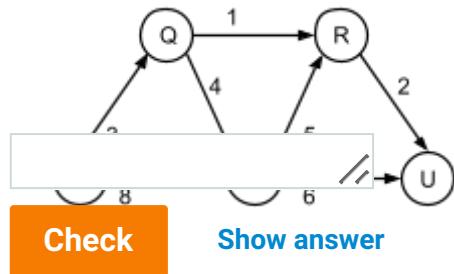


//
Check**Show answer**

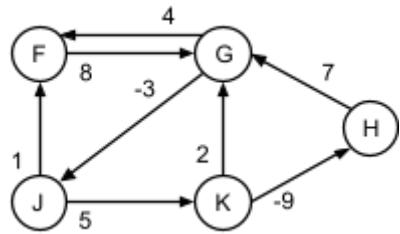
- 3) The shortest path from S to U has a length of ____.

©zyBooks 07/19/23 22:03 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023





- 4) Given a path H, G, J, F, the path length is ____.



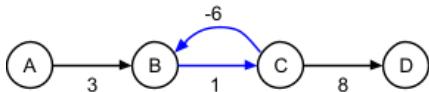
Check

Show answer

Negative edge weight cycles

The **cycle length** is the sum of the edge weights in a cycle. A **negative edge weight cycle** has a cycle length less than 0. A shortest path does not exist in a graph with a negative edge weight cycle, because each loop around the negative edge weight cycle further decreases the cycle length, so no minimum exists.

Figure 16.8.1: A shortest path from A to D does not exist, because the cycle B, C can be repeatedly taken to further reduce the cycle length.



Path 1: A, B, C, B, C, D
 Path 2: A, B, C, B, C, B, C, D
 Path 3: A, B, C, B, C, B, C, B, C, D
and so on...

Length = $3 + 1 + -6 + 1 + 8 = 7$
 Length = $3 + 1 + -6 + 1 + -6 + 1 + 8 = 2$
 Length = $3 + 1 + -6 + 1 + -6 + 1 + -6 + 1 + 8 = -3$

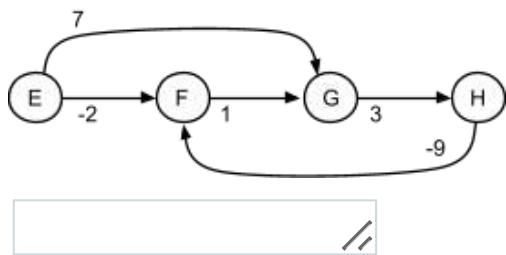
PARTICIPATION ACTIVITY
16.8.5: Negative edge weight cycles.

©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

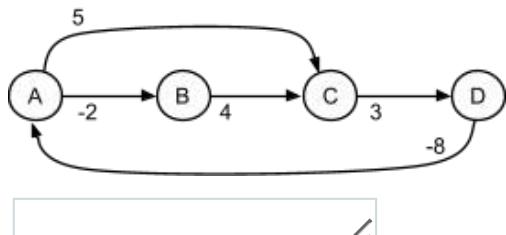
COLORADOCSPB2270Summer2023

- 1) The cycle length for F, G, H, F is _____.



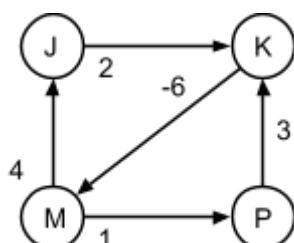
[Show answer](#)


- 2) Is A, C, D, A a negative edge weight cycle? Type Yes or No.



[Show answer](#)


- 3) The graph contains _____ negative edge weight cycles.



©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



Check**Show answer****CHALLENGE ACTIVITY****16.8.1: Weighted graphs.**

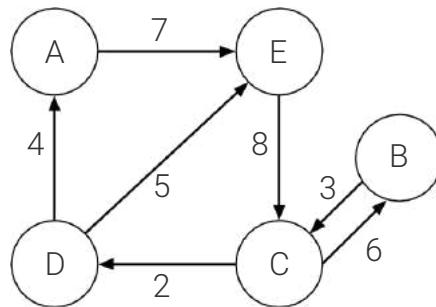
489394.3384924.qx3zqy7

Start

©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



Find the weight of each of the following edges.

Weight of edge from C to B: Enter "undefined" if the edge is not defined.Weight of edge from A to D: Weight of edge from C to D:

1

2

3

4

5

Check**Next**

16.9 Algorithm: Dijkstra's shortest path

©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Dijkstra's shortest path algorithm

Finding the shortest path between vertices in a graph has many applications. Ex: Finding the shortest driving route between two intersections can be solved by finding the shortest path in a directed graph where vertices are intersections and edge weights are distances. If edge weights instead are expected travel times (possibly based on real-time traffic data), finding the shortest path will provide the fastest driving route.

Dijkstra's shortest path algorithm, created by Edsger Dijkstra, determines the shortest path from a start vertex to each vertex in a graph. For each vertex, Dijkstra's algorithm determines the vertex's distance and predecessor pointer. A vertex's **distance** is the shortest path distance from the start vertex. A vertex's **predecessor pointer** points to the previous vertex along the shortest path from the start vertex.

Dijkstra's algorithm initializes all vertices' distances to infinity (∞), initializes all vertices' predecessors to null, and enqueues all vertices into a queue of unvisited vertices. The algorithm then assigns the start vertex's distance with 0. While the queue is not empty, the algorithm dequeues the vertex with the shortest distance. For each adjacent vertex, the algorithm computes the distance of the path from the start vertex to the current vertex and continuing on to the adjacent vertex. If that path's distance is shorter than the adjacent vertex's current distance, a shorter path has been found. The adjacent vertex's current distance is updated to the distance of the newly found shorter path's distance, and vertex's predecessor pointer is pointed to the current vertex.

PARTICIPATION ACTIVITY

16.9.1: Dijkstra's algorithm finds the shortest path from a start vertex to each vertex in a graph.

**Animation content:**

undefined

Animation captions:

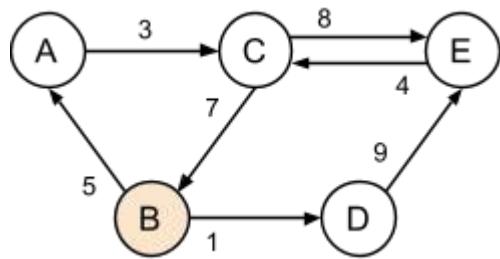
1. Each vertex is initialized with distance set to Infinity and the predecessor pointer set to null.
Each vertex is enqueued into unvisitedQueue.
2. The start vertex's distance is 0. The algorithm visits the start vertex first.
3. For each adjacent vertex, if a shorter path from the start vertex to the adjacent vertex is found, the vertex's distance and predecessor pointer are updated.
4. B has the shortest path distance, and is dequeued from the queue. The path through B to C is not shorter, so no update occurs. The path through B to D is shorter, so D's distance and predecessor are updated.
5. D is then dequeued. The path through D to C is shorter, so C's distance and predecessor pointer are updated.
6. C is then dequeued. The path through C to D is not shorter, so no update occurs.
7. The algorithm terminates when all vertices are visited. Each vertex's distance is the shortest path distance from the start vertex. The vertex's predecessor pointer points to the previous vertex in the shortest path.

PARTICIPATION ACTIVITY

16.9.2: Dijkstra's shortest path traversal.



Perform Dijkstra's shortest path algorithm on the graph below with starting vertex B.



- 1) Which vertex is visited first?

Check**Show answer**

©zyBooks 07/19/23 22:03 169246
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 2) A's distance after the algorithm's first while loop iteration is ____.
Type inf for infinity.

Check**Show answer**

- 3) D's distance after the first while loop iteration is ____.
Type inf for infinity.

Check**Show answer**

- 4) C's distance after the first iteration of the while loop is ____.
Type inf for infinity.

Check**Show answer**

- 5) Which vertex is visited second?

Check**Show answer**

©zyBooks 07/19/23 22:03 169246
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 6) E's distance after the second while loop iteration is ____.



**Check****Show answer**

©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Finding shortest path from start vertex to destination vertex

After running Dijkstra's algorithm, the shortest path from the start vertex to a destination vertex can be determined using the vertices' predecessor pointers. If the destination vertex's predecessor pointer is not 0, the shortest path is traversed in reverse by following the predecessor pointers until the start vertex is reached. If the destination vertex's predecessor pointer is null, then a path from the start vertex to the destination vertex does not exist.

PARTICIPATION ACTIVITY

16.9.3: Determining the shortest path from Dijkstra's algorithm.



Animation content:

undefined

Animation captions:

1. The vertex's predecessor pointer points to the previous vertex in the shortest path.
2. Starting with the destination vertex, the predecessor pointer is followed until the start vertex is reached.
3. The vertex's distance is the shortest path distance from the start vertex.

PARTICIPATION ACTIVITY

16.9.4: Shortest path based on vertex predecessor.



Type path as: A, B, C

If path does not exist, type: None

©zyBooks 07/19/23 22:03 1692462

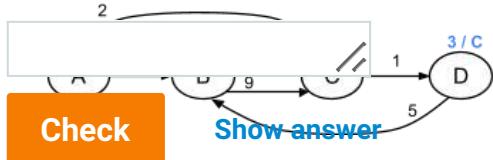
Taylor Larrechea

COLORADOCSPB2270Summer2023

1) After executing

DijkstraShortestPath(A), what is the shortest path from A to B?





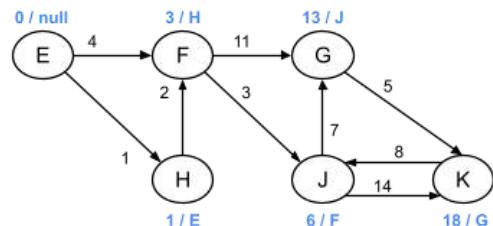
©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

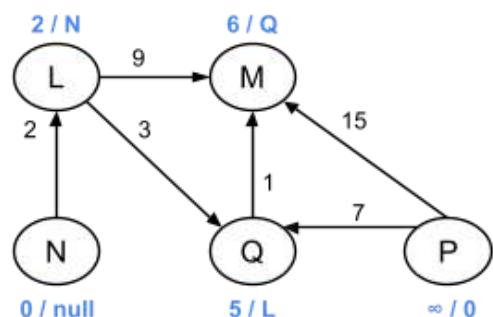
2) After executing

DijkstraShortestPath(E), what is the shortest path from E to G?

**Check****Show answer**

3) After executing

DijkstraShortestPath(N), what is the shortest path from N to P?

**Check****Show answer**

©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Algorithm efficiency

If the unvisited vertex queue is implemented using a list, the runtime for Dijkstra's shortest path algorithm is $O(V^2)$. The outer loop executes V times to visit all vertices. In each outer loop execution, dequeuing the vertex from the queue requires searching all vertices in the list, which has a runtime of $O(V)$. For each vertex, the algorithm follows the subset of edges to adjacent vertices; following a total of E edges across all loop executions. Given $E < V^2$, the runtime is $O(V^2 + E) = O(V^2 + E) = O(V^2)$. Implementing the unvisited vertex queue using a standard binary heap reduces the runtime to $O((E + V) \log V)$, and using a Fibonacci heap data structure (not discussed in this material) reduces the runtime to $O(E + V \log V)$.

Negative edge weights

Dijkstra's shortest path algorithm can be used for unweighted graphs (using a uniform edge weight of 1) and weighted graphs with non-negative edges weights. For a directed graph with negative edge weights, Dijkstra's algorithm may not find the shortest path for some vertices, so the algorithm should not be used if a negative edge weight exists.

PARTICIPATION ACTIVITY

16.9.5: Dijkstra's algorithm may not find the shortest path for a graph with negative edge weights.



Animation content:

undefined

Animation captions:

1. A is the start vertex. Adjacent vertices resulting in a shorter path are updated.
2. Path through B to D results in a shorter path. Vertex D is updated.
3. D has no adjacent vertices.
4. A path through C to B results in a shorter path. Vertex B is updated.
5. Vertex B has already been visited, and will not be visited again. So, D's distance and predecessor are not for the shortest path from A to D.

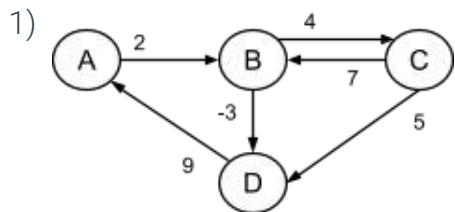
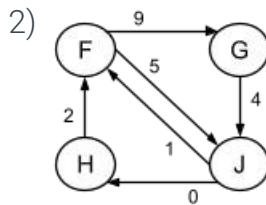
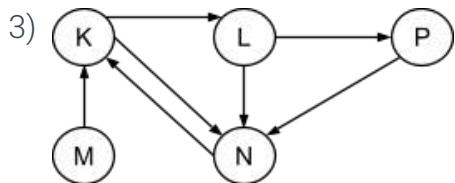
©zyBooks 07/19/23 22:03 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

16.9.6: Dijkstra's shortest path algorithm: Supported graph types.



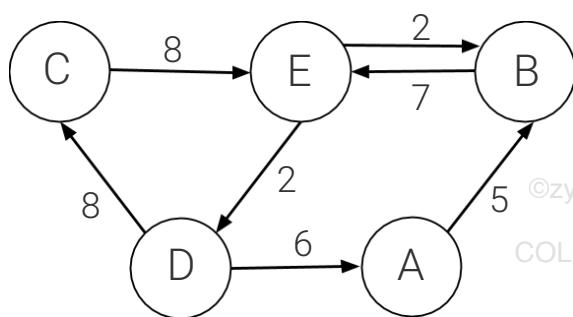
Indicate if Dijkstra's algorithm will find the shortest path for the following graphs.

 Yes No Yes No Yes No
CHALLENGE ACTIVITY

16.9.1: Dijkstra's shortest path.



489394.3384924.qx3zqy7

Start

©zyBooks 07/19/23 22:03 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Starting at vertex B, what is the shortest path length to each vertex?

A: Ex: 1

B:

C:
D:
E:

1

2

3

[Check](#)[Next](#)

©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

16.10 Algorithm: Bellman-Ford's shortest path



This section has been set as optional by your instructor.

Bellman-Ford shortest path algorithm

The **Bellman-Ford shortest path algorithm**, created by Richard Bellman and Lester Ford, Jr., determines the shortest path from a start vertex to each vertex in a graph. For each vertex, the Bellman-Ford algorithm determines the vertex's distance and predecessor pointer. A vertex's **distance** is the shortest path distance from the start vertex. A vertex's **predecessor pointer** points to the previous vertex along the shortest path from the start vertex.

The Bellman-Ford algorithm initializes all vertices' current distances to infinity (∞) and predecessors to null, and assigns the start vertex with a distance of 0. The algorithm performs $V-1$ main iterations, visiting all vertices in the graph during each iteration. Each time a vertex is visited, the algorithm follows all edges to adjacent vertices. For each adjacent vertex, the algorithm computes the distance of the path from the start vertex to the current vertex and continuing on to the adjacent vertex. If that path's distance is shorter than the adjacent vertex's current distance, a shorter path has been found. The adjacent vertex's current distance is updated to the newly found shorter path's distance, and the vertex's predecessor pointer is pointed to the current vertex.

The Bellman-Ford algorithm does not require a specific order for visiting vertices during each main iteration. So after each iteration, a vertex's current distance and predecessor may not yet be the shortest path from the start vertex. The shortest path may propagate to only one vertex each iteration, requiring $V-1$ iterations to propagate from the start vertex to all other vertices.

PARTICIPATION
ACTIVITY

16.10.1: The Bellman-Ford algorithm finds the shortest path from a source vertex to all other vertices in a graph.



Animation content:

undefined

Animation captions:

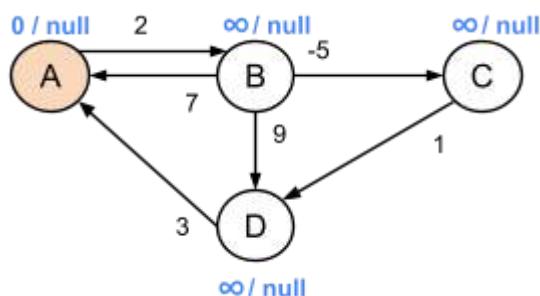
1. The Bellman-Ford algorithm initializes each vertex's distance to Infinity and each vertex's predecessor to null, and assigns the start vertex's distance with 0.
2. For each vertex in the graph, if a shorter path from the currentV to the adjacent vertex is found, the adjacent vertex's distance and predecessor pointer are updated. ©zyBooks 07/19/23 22:03 1692462 Taylor Larrechea
3. The path through B to D results in a shorter path to D than is currently known. So vertex D is updated.
4. The path through C to B results in a shorter path to B than is currently known. So vertex B is updated.
5. D has no adjacent vertices. After each iteration, a vertex's distance and predecessor may not yet be the shortest path from the start vertex.
6. In each main iteration, the algorithm visits all vertices. This time, the path through B to D results in an even shorter path. So vertex D is updated again.
7. During the third main iteration, no shorter paths are found, so no vertices are updated. V-1 iterations may be required to propagate the shortest path from the start vertex to all other vertices.
8. When done, each vertex's distance is the shortest path distance from the start vertex, and the vertex's predecessor pointer points to the previous vertex in the shortest path.

PARTICIPATION ACTIVITY

16.10.2: Bellman-Ford shortest path traversal.



The start vertex is A. In each main iteration, vertices in the graph are visited in the following order: A, B, C, D.



- 1) What are B's values after the first iteration?

- 2 / A
- ∞ / null

- 2) What are C's values after the first iteration?

©zyBooks 07/19/23 22:03 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



- 5 / B
- 3 / B
- ∞ / null

3) What are D's values after the first iteration?

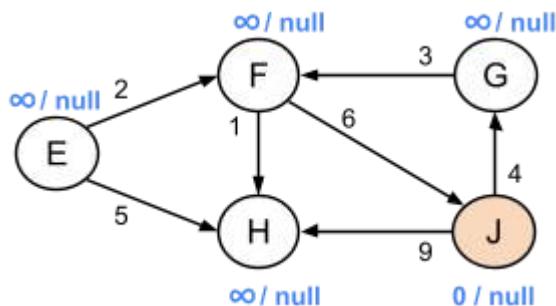
- 3 / A
- 2 / C
- ∞ / null

©zyBooks 07/19/23 22:03 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

16.10.3: Bellman-Ford: Distance and predecessor values.

The start vertex is J. Vertices in the graph are processed in the following order: E, F, G, H, J.



1) How many iterations are executed?

- 5
- 4
- 6

2) What are F's values after the first iteration?

- ∞ / null
- 7 / G

©zyBooks 07/19/23 22:03 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

3) What are G's values after the first iteration?

- 4 / J
- ∞ / null



- 4) What are E's values after the final iteration?

- 9 / F
- ∞ / null

Algorithm Efficiency

©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

The runtime for the Bellman-Ford shortest path algorithm is $O(VE)$. The outer loop (the main iterations) executes $V-1$ times. In each outer loop execution, the algorithm visits each vertex and follows the subset of edges to adjacent vertices, following a total of E edges across all loop executions.

Checking for negative edge weight cycles

The Bellman-Ford algorithm supports graphs with negative edge weights. However, if a negative edge weight cycle exists, a shortest path does not exist. After visiting all vertices $V-1$ times, the algorithm checks for negative edge weight cycles. If a negative edge weight cycle does not exist, the algorithm returns true (shortest path exists), otherwise returns false.

PARTICIPATION ACTIVITY

16.10.4: Bellman-Ford: Checking for negative edge weight cycles.



Animation content:

undefined

Animation captions:

1. After visiting all vertices $V-1$ times, the Bellman-Ford algorithm checks for negative edge weight cycles.
2. For each vertex in the graph, adjacent vertices are checked for a shorter path. No such path exists through A to B.
3. But, a shorter path is still found through B to C, so a negative edge weight cycle exists.
4. The algorithm returns false, indicating a shortest path does not exist.

©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Figure 16.10.1: Bellman-Ford shortest path algorithm.

```

BellmanFord(startV) {
    for each vertex currentV in graph {
        currentV->distance = Infinity
        currentV->predV = null
    }

    // startV has a distance of 0 from itself
    startV->distance = 0

    for i = 1 to number of vertices - 1 { // Main iterations
        for each vertex currentV in graph {
            for each vertex adjV adjacent to currentV {
                edgeWeight = weight of edge from currentV to adjV
                alternativePathDistance = currentV->distance +
edgeWeight

                // If shorter path from startV to adjV is found,
                // update adjV's distance and predecessor
                if (alternativePathDistance < adjV->distance) {
                    adjV->distance = alternativePathDistance
                    adjV->predV = currentV
                }
            }
        }
    }

    // Check for a negative edge weight cycle
    for each vertex currentV in graph {
        for each vertex adjV adjacent to currentV {
            edgeWeight = weight of edge from currentV to adjV
            alternativePathDistance = currentV->distance +
edgeWeight

            // If shorter path from startV to adjV is still found,
            // a negative edge weight cycle exists
            if (alternativePathDistance < adjV->distance) {
                return false
            }
        }
    }

    return true
}

```

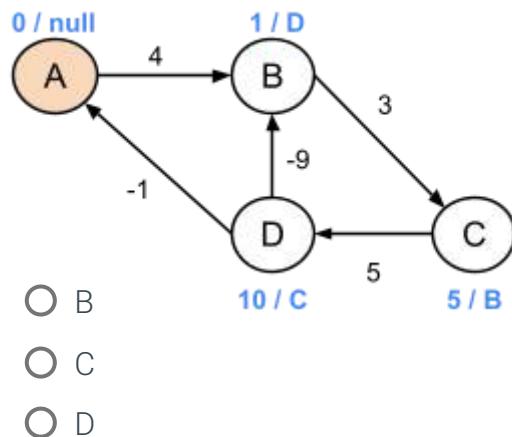
©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea
COLORADOCSPB2270Summer2023**PARTICIPATION ACTIVITY****16.10.5: Bellman-Ford algorithm: Checking for negative edge weight cycles.**

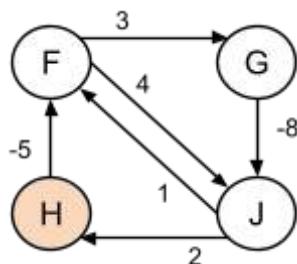
- 1) Given the following result from the Bellman-Ford algorithm for a start vertex of A, a negative edge weight cycle is found when checking adjacent



vertices of vertex ____.



- 2) What does the Bellman-Ford algorithm return for the following graph with a start vertex of H?

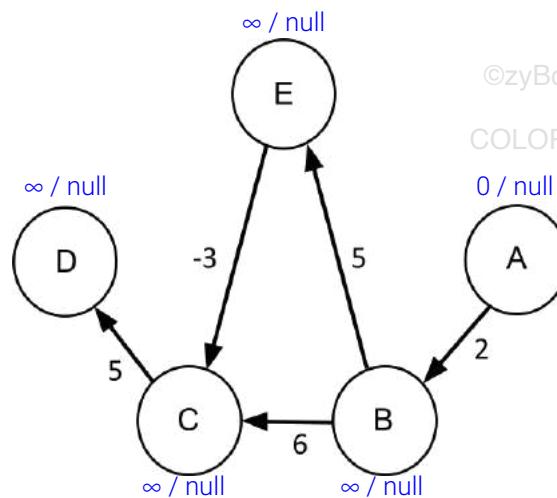


CHALLENGE ACTIVITY

16.10.1: Bellman-Ford's shortest path.

Start

The Bellman-Ford algorithm is run on the following graph. The start vertex is A. Assume each of the algorithm visits vertices in the following order: A, B, C, D, E.



©zyBooks 07/19/23 22:03 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

How many main loop iterations are executed? Ex: 1

What are B's values after the first iteration? Ex: 0 or inf / ✓ Enter inf for ∞.

What are C's values after the first iteration? / ✓

What are D's values after the first iteration? / ✓

What are E's values after the first iteration? / ✓

1

2

3

4

Check**Next**

16.11 Topological sort

©zyBooks 07/19/23 22:03 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Overview

A **topological sort** of a directed, acyclic graph produces a list of the graph's vertices such that for every edge from a vertex X to a vertex Y, X comes before Y in the list.

PARTICIPATION ACTIVITY

16.11.1: Topological sort.

**Animation captions:**

1. Analysis of each edge in the graph determines if an ordering of vertices is a valid topological sort.
2. If an edge from X to Y exists, X must appear before Y in a valid topological sort. C, D, A, F, B, E is not valid because this requirement is violated for three edges.
©zyBooks 07/19/23 22:03 1692462
Taylor Larrochea
COLORADOCSPB2270Summer2023
3. Ordering D, A, F, E, C, B has 1 edge violating the requirement, so the ordering is not a valid topological sort.
4. For ordering D, A, F, E, B, C, the requirement holds for all edges, so the ordering is a valid topological sort.
5. A graph can have more than 1 valid topological sort. Another valid ordering is D, A, F, B, E, C.

PARTICIPATION ACTIVITY

16.11.2: Topological sort.



1) In the example above, D, A, B, F, E, C is a valid topological sort.

- True
- False

2) Which of the following is NOT a requirement of the graph for topological sorting?

- The graph must be acyclic.
- The graph must be directed.
- The graph must be weighted.

3) For a directed, acyclic graph, only one possible topological sort output exists.

- True
- False

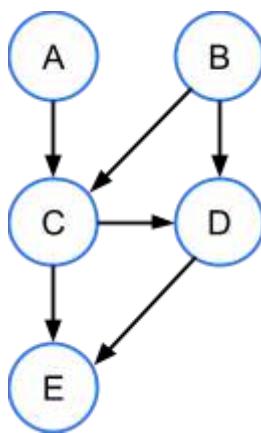
©zyBooks 07/19/23 22:03 1692462
Taylor Larrochea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

16.11.3: Identifying valid topological sorts.



Indicate whether each vertex ordering is a valid topological sort of the graph below.



©zyBooks 07/19/23 22:03 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 1) A, B, C, D, E □

- Valid
 Invalid

- 2) E, D, C, B, A □

- Valid
 Invalid

- 3) D, E, A, B, C □

- Valid
 Invalid

- 4) B, A, C, D, E □

- Valid
 Invalid

- 5) B, A, C, E, D □

- Valid
 Invalid

Example: course prerequisites

©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea
COLORADOCSPB2270Summer2023

Graphs can be used to indicate a sequence of steps, where an edge from X to Y indicates that X must be done before Y. A topological sort of such a graph provides one possible ordering for performing the steps. Ex: Given a graph representing course prerequisites, a topological sort of the graph provides an ordering in which the courses can be taken.

PARTICIPATION
ACTIVITY

16.11.4: Topological sorting can be used to order course prerequisites. □

Animation captions:

1. For a graph representing course prerequisites, the vertices represent courses, and the edges represent the prerequisites. CS 101 must be taken before CS 102, and CS 102 before CS 103.
2. CS 103 is "Robotics Programming" and has a physics course (Phys 101) as a prerequisite. Phys 101 also has a math prerequisite (Math 101).
3. The graph's valid topological sorts provide possible orders in which to take the courses.

©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

16.11.5: Course prerequisites.



1) The "Math 101" and "CS 101" vertices have no incoming edges, and therefore one of these two vertices must be the first vertex in any topological sort.

- True
 False



2) Every topological sort ends with the "CS 103" vertex because this vertex has no outgoing edges.

- True
 False



Topological sort algorithm

The topological sort algorithm uses three lists: a results list that will contain a topological sort of vertices, a no-incoming-edges list of vertices with no incoming edges, and a remaining-edges list. The result list starts as an empty list of vertices. The no-incoming-edges vertex list starts as a list of all vertices in the graph with no incoming edges. The remaining-edges list starts as a list of all edges in the graph.

The algorithm executes while the no-incoming-edges vertex list is not empty. For each iteration, a vertex is removed from the no-incoming-edges list and added to the result list. Next, a temporary list is built by removing all edges in the remaining-edges list that are outgoing from the removed vertex. For each edge currentE in the temporary list, the number of edges in the remaining-edges list that are incoming to currentE's terminating vertex are counted. If the incoming edge count is 0, then currentE's terminating vertex is added to the no-incoming-edges vertex list.

Because each loop iteration can remove any vertex from the no-incoming-edges list, the algorithm's output is not guaranteed to be the graph's only possible topological sort.

**Animation content:**

undefined

Animation captions:

©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

1. The topological sort algorithm begins by initializing an empty result list, a list of all vertices with no incoming edges, and a "remaining edges" list with all edges in the graph.
2. Vertex E is removed from the list of vertices with no incoming edges and added to resultList. Outgoing edges from E are removed from remainingEdges and added to outgoingEdges.
3. Edge EF goes to vertex F, which still has 2 incoming edges. Edge EG goes to vertex G, which still has 1 incoming edge.
4. Vertex A is removed and added to resultList. Outgoing edges from A are removed from remainingEdges. Vertices B and C are added to noIncoming.
5. Vertices C and D are processed, each with 1 outgoing edge.
6. Vertices B and F are processed, each also with 1 outgoing edge.
7. Vertex G is processed last. No outgoing edges remain. The final result is E, A, C, D, B, F, G.

Figure 16.11.1: GraphGetIncomingEdgeCount function.

```
GraphGetIncomingEdgeCount(edgeList,  
vertex) {  
    count = 0  
    for each edge currentE in edgeList {  
        if (edge->toVertex == vertex) {  
            count = count + 1  
        }  
    }  
    return count  
}
```

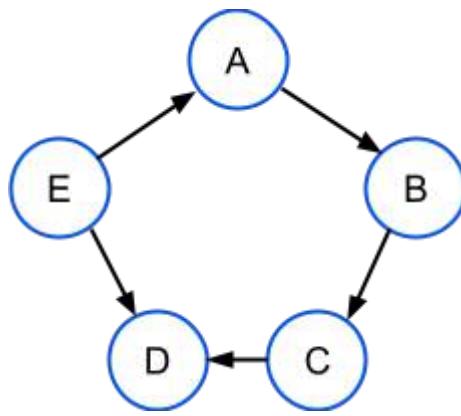
©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



Consider calling GraphTopologicalSort on the graph below.



©zyBooks 07/19/23 22:03 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 1) In the first iteration of the while loop,
what is assigned to currentV?

- Vertex A
- Vertex E
- Undefined

- 2) In the first iteration of the while loop,
what is the contents of outgoingEdges
right before the for-each loop begins?

- Edge from E to A and edge from
E to D
- Edge from A to B
- No edges

- 3) When currentV becomes vertex C, what
is the contents of resultList?

- A, B
- E, A, B
- E, D

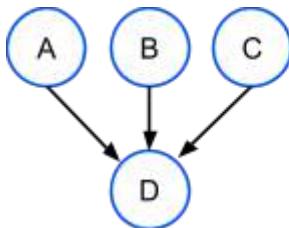
- 4) What is the final contents of resultList?

- A, B, C, D, E
- E, A, B, C, D
- E, A, B, D, C

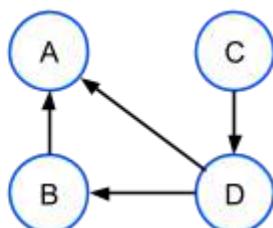
©zyBooks 07/19/23 22:03 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION
ACTIVITY

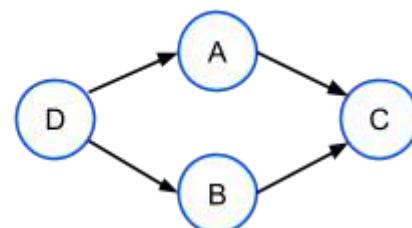
16.11.8: Topological sort matching.



1



2



©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

3

If unable to drag and drop, refresh the page.

Graph 1**Graph 3****Graph 2**

D, B, A, C

C, D, B, A

B, C, A, D

Reset**PARTICIPATION ACTIVITY**

16.11.9: Topological sort algorithm.



1) What does `GraphTopologicalSort` return?



- A list of vertices.
- A list of edges.
- A list of indices.

2) `GraphTopologicalSort` will not work on a graph with a positive number of vertices but no edges.



- True
- False

3) If a graph implementation stores incoming and outgoing edge counts in each vertex, then the statement
`GraphGetIncomingEdgeCount(remainingEdges,`



©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

`edge->to`) can be replaced with
`currentE->toVertex->incomingEdgeCount`.

- True
- False

©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Algorithm efficiency

The two vertex lists used in the topological sort algorithm will at most contain all the vertices in the graph. The remaining-edge list will at most contain all edges in the graph. Therefore, for a graph with a set of vertices V and a set of edges E , the space complexity of topological sorting is $O(|V| + |E|)$. If a graph implementation allows for retrieval of a vertex's incoming and outgoing edges in constant time, then the time complexity of topological sorting is also $O(|V| + |E|)$.

16.12 Minimum spanning tree

Overview

A graph's **minimum spanning tree** is a subset of the graph's edges that connect all vertices in the graph together with the minimum sum of edge weights. The graph must be weighted and connected. A **connected** graph contains a path between every pair of vertices.

PARTICIPATION ACTIVITY

16.12.1: Using the minimum spanning to minimize total length of power lines connecting cities.

©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Animation captions:

1. A minimum spanning tree can be used to find the minimal amount of power lines needed to connect cities. Each vertex represents a city. Edges represent roads between cities. The city P has a power plant.

2. Power lines are along roads, such that each city is connected to a powered city. But power lines along every road would be excessive.
3. The minimum spanning tree, shown in red, is the set of edges that connect all cities to power with minimal total power line length.
4. The resulting minimum spanning tree can be viewed as a tree with the power plant city as the root.

©zyBooks 07/19/23 22:03 1692462

PARTICIPATION ACTIVITY

16.12.2: Minimum spanning tree.

Taylor Larrechea
COLORADOCSPB2270Summer2023

1) If no path exists between 2 vertices in a weighted and undirected graph, then no minimum spanning tree exists for the graph.

- True
 False

2) A minimum spanning tree is a set of vertices.

- True
 False

3) The "minimum" in "minimum spanning tree" refers to the sum of edge weights.

- True
 False

4) A minimum spanning tree can only be built for an undirected graph.

- True
 False



Kruskal's minimum spanning tree algorithm

©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

Kruskal's minimum spanning tree algorithm determines subset of the graph's edges that connect all vertices in an undirected graph with the minimum sum of edge weights. Kruskal's minimum spanning tree algorithm uses 3 collections:

- An edge list initialized with all edges in the graph.
- A collection of vertex sets that represent the subsets of vertices connected by current set of edges in the minimum spanning tree. Initially, the vertex sets consists of one set for each vertex.

- A set of edges forming the resulting minimum spanning tree.

The algorithm executes while the collection of vertex sets has at least 2 sets and the edge list has at least 1 edge. In each iteration, the edge with the lowest weight is removed from the list of edges. If the removed edge connects two different vertex sets, then the edge is added to the resulting minimum spanning tree, and the two vertex sets are merged.

PARTICIPATION ACTIVITY

16.12.3: Minimum spanning tree algorithm.

 ©zyBooks 07/19/23 22:03 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

Animation content:

undefined

Animation captions:

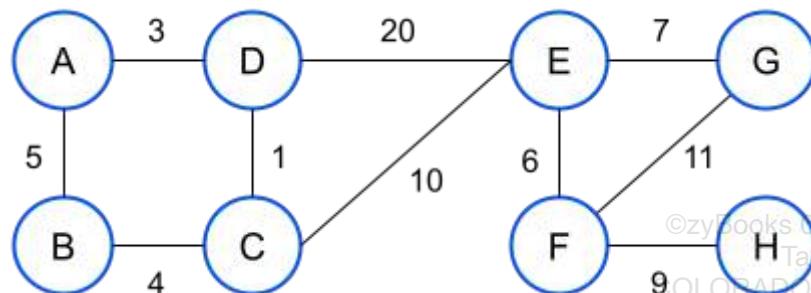
1. An edge list, a collection of vertex sets, and an empty result set are initialized. The edge list contains all edges from the graph.
2. Edge AD is removed from the edge list and added to resultList, which will contain the edges forming the minimum spanning tree.
3. The next 5 edges connect different vertex sets and are added to the result.
4. Edges AB and CE both connect 2 vertices that are in the same vertex set, and therefore are not added to the result.
5. Edge EF connects the 2 remaining vertex sets.
6. One vertex set remains, so the minimum spanning tree is complete.

PARTICIPATION ACTIVITY

16.12.4: Minimum spanning tree algorithm.



Consider executing Kruskal's minimum spanning tree algorithm on the following graph:


 ©zyBooks 07/19/23 22:03 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

- 1) What is the first edge that will be added to the result?

- AD
- AB



- BC
- CD

2) What is the second edge that will be added to the result?



- AD
- AB
- BC
- CD

3) What is the first edge that will NOT be added to the result?



- BC
- AB
- FG
- DE

4) How many edges will be in the resulting minimum spanning tree?



- 5
- 7
- 9
- 10

PARTICIPATION ACTIVITY

16.12.5: Minimum spanning tree - critical thinking.



1) The edge with the lowest weight will always be in the minimum spanning tree.



- True
- False

2) The minimum spanning tree may contain all edges from the graph.



- True
- False

©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



3) Only 1 minimum spanning tree exists for a graph that has no duplicate edge weights.

- True
- False

4) The edges from any minimum spanning tree can be used to create a path that goes through all vertices in the graph without ever encountering the same vertex twice.

- True
- False

©zyBooks 07/19/23 22:03 169246
Taylor Larrechea
COLORADOCSPB2270Summer2023

Algorithm efficiency

Kruskal's minimum spanning tree algorithm's use of the edge list, collection of vertex sets, and resulting edge list results in a space complexity of . If the edge list is sorted at the beginning, then the minimum edge can be removed in constant time within the loop. Combined with a mechanism to map a vertex to the containing vertex set in constant time, the minimum spanning tree algorithm has a runtime complexity of .

CHALLENGE ACTIVITY

16.12.1: Minimum spanning tree.

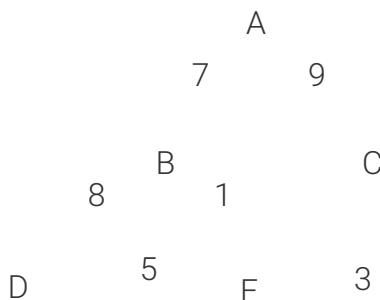


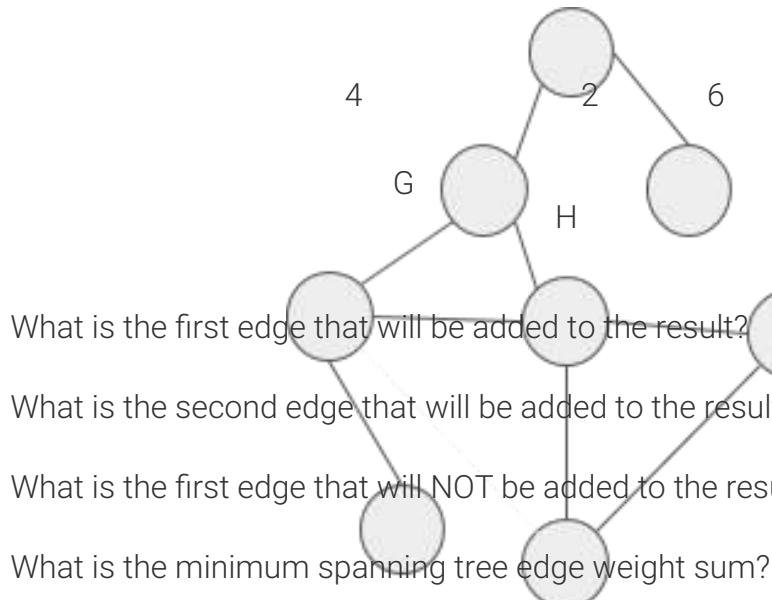
489394.3384924.qx3zqy7

Start

Kruskal's minimum spanning tree algorithm is executed on the following graph.

©zyBooks 07/19/23 22:03 169246
Taylor Larrechea
COLORADOCSPB2270Summer2023





Check**Next**

16.13 All pairs shortest path



This section has been set as optional by your instructor.

Overview and shortest paths matrix

An **all pairs shortest path** algorithm determines the shortest path between all possible pairs of vertices in a graph. For a graph with vertices V , a $|V| \times |V|$ matrix represents the shortest path lengths between all vertex pairs in the graph. Each row corresponds to a start vertex, and each column in the matrix corresponds to a terminating vertex for each path. Ex: The matrix entry at row F and column T represents the shortest path length from vertex F to vertex T.

PARTICIPATION ACTIVITY

16.13.1: Shortest path lengths matrix.

©zyBooks 07/19/23 22:03 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Animation captions:

1. For a graph with 4 vertices, the all-pairs-shortest-path matrix has 4 rows and 4 columns. An entry exists for every possible vertex pair.
2. An entry at row F and column T represents the shortest path length from vertex F to vertex T.

3. The shortest path from vertex A to vertex D has length 9.
4. The shortest path from B to A has length 2.
5. Only total path lengths are stored in the matrix, not the actual sequence of edges for the corresponding path.

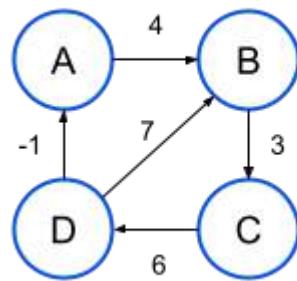
PARTICIPATION ACTIVITY

16.13.2: Shortest path lengths matrix.

©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



	A	B	C	D
A	0	4	7	?
B	8	0	3	9
C	5	9	?	6
D	-1	?	6	0

- 1) What is the shortest path length from A to D?

//
Check**Show answer**

- 2) What is the shortest path length from C to C?

//
Check**Show answer**

- 3) What is the shortest path length from D to B?

//
Check**Show answer**

©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

16.13.3: Shortest path lengths matrix.

- 1) If a graph has 11 vertices and 7 edges, how many entries are in the all-pairs-

shortest-path matrix?

©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

- 11
- 7
- 77
- 121

2) An entry at row R and column C in the matrix represents the shortest path length from vertex C to vertex R.



- True
- False

3) If a graph contains a negative edge weight, the matrix of shortest path lengths will contain at least 1 negative value.



- True
- False

4) For a matrix entry representing path length from A to B, when no such path exists, a special-case value must be used to indicate that no path exists.



- True
- False

Floyd-Warshall algorithm

©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

The **Floyd-Warshall all-pairs shortest path algorithm** generates a $|V| \times |V|$ matrix of values representing the shortest path lengths between all vertex pairs in a graph. Graphs with cycles and negative edge weights are supported, but the graph must not have any negative cycles. A **negative cycle** is a cycle with edge weights that sum to a negative value. Because a negative cycle could be traversed repeatedly, lowering the path length each time, determining a shortest path between 2 vertices in a negative cycle is not possible.

The Floyd-Warshall algorithm initializes the shortest path lengths matrix in 3 steps.

1. Every entry is assigned with infinity.
2. Each entry representing the path from a vertex to itself is assigned with 0.
3. For each edge from X to Y in the graph, the matrix entry for the path from X to Y is initialized with the edge's weight.

The algorithm then iterates through every vertex in the graph. For each vertex X, the shortest path lengths for all vertex pairs are recomputed by considering vertex X as an intermediate vertex. For each matrix entry representing A to B, existing matrix entries are used to compute the length of the path from A through X to B. If this path length is less than the current shortest path length, then the corresponding matrix entry is updated.

PARTICIPATION ACTIVITY**16.13.4: Floyd-Warshall algorithm.****Animation content:**

undefined

Animation captions:

1. All entries in the shortest path lengths matrix are first initialized with ∞ . Vertex-to-same-vertex values are then initialized with 0. For each edge, the corresponding entry for from X to Y is initialized with the edge's weight.
2. The $k = 0$ iteration corresponds to vertex A. Each vertex pair X, Y is analyzed to see if the path from X through A to Y yields a shorter path. Shorter paths from C to B and from D to B are found.
3. During the $k = 1$ iteration, 4 shorter paths that pass through vertex B are found from path from A to C, from A to D, from C to D, and from D to C.
4. During the $k = 2$ iteration, 1 shorter path that passes through vertex C is found from the path from B to A.
5. During the $k=3$ iteration, no entries are updated. The shortest path lengths matrix is complete.

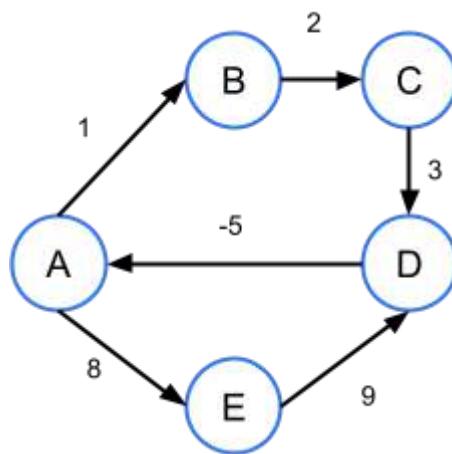
PARTICIPATION ACTIVITY**16.13.5: Floyd-Warshall algorithm.**

©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADO CSPB2270Summer2023

Consider executing the Floyd-Warshall algorithm on the graph below.



©zyBooks 07/19/23 22:03 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 1) How many rows and columns are in the shortest path length matrix? □
 - 5 rows, 5 columns
 - 5 rows, 6 columns
 - 6 rows, 5 columns
 - 6 rows, 6 columns

- 2) After matrix initialization, but before the k-loop starts, how many entries are set to non-infinite values? □
 - 5
 - 11
 - 14
 - 25

- 3) After the algorithm completes, how many entries in the matrix are negative values? □
 - 0
 - 3
 - 5
 - 7

- 4) After the algorithm completes, what is the shortest path length from vertex B to vertex E? □
 - 4
 - 5

©zyBooks 07/19/23 22:03 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 8
- Infinity (no path)

Path reconstruction

©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea
COLORADOCSPB2270Summer2023

Although only shortest path lengths are computed by the Floyd-Warshall algorithm, the matrix can be used to reconstruct the path sequence. Given the shortest path length from a start vertex to an end vertex is L. An edge from vertex X to the ending vertex exists such that the shortest path length from the starting vertex to X, plus the edge weight, equals L. Each such edge is found, and the path is reconstructed in reverse order.

PARTICIPATION ACTIVITY

16.13.6: Path reconstruction.



Animation captions:

1. The matrix built by the Floyd-Warshall algorithm indicates that the shortest path from A to C is of length 3.
2. The path from A to C is determined by backtracking from C.
3. Since A is the path's starting point, shortest distances from A are relevant at each vertex.
4. Traversing backwards along the only incoming edge to C, the expected path length at B is $3 - 5 = -2$.
5. The computation at the edge from A to B doesn't hold, so the edge is not part of the path.
6. The computation on the D to B edge holds.
7. The computation on the A to D edge holds, and brings the path back to the starting vertex. The final path is A to D to B to C.

Figure 16.13.1: FloydWarshallReconstructPath algorithm.

©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea
COLORADOCSPB2270Summer2023

```
FloydWarshallReconstructPath(graph, startVertex, endVertex,
distMatrix) {
    path = new, empty path

    // Backtrack from the ending vertex
    currentV = endVertex
    while (currentV != startVertex) {
        incomingEdges = all edges in the graph incoming to current
        vertex
        for each edge currentE in incomingEdges {
            expected = distMatrix[startVertex][currentV] -
            currentE->weight
            actual = distMatrix[startVertex][currentE->fromVertex]
            if (expected == actual) {
                currentV = currentE->fromVertex
                Prepend currentE to path
                break
            }
        }
    }

    return path
}
```

©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

PARTICIPATION
ACTIVITY

16.13.7: Path reconstruction.



1) Path reconstruction from vertex X to vertex Y is only possible if the matrix entry is non-infinite.

- True
- False



2) A path with positive length may include edges with negative weights.

- True
- False



3) More than 1 possible path sequence from vertex X to vertex Y may exist, even though the matrix will only store 1 path length from X to Y.

- True
- False

©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



- 4) Path reconstruction is not possible if the graph has a cycle.

- True
- False

Algorithm efficiency

©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

The Floyd-Warshall algorithm builds a $|V| \times |V|$ matrix and therefore has a space complexity of $\Theta(|V|^3)$. The matrix is constructed with a runtime complexity of $\Theta(|V|^3)$.

©zyBooks 07/19/23 22:03 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

17.1 Time-ordered behavior

Time-ordered behavior is system functionality where outputs depend on the order in which input events occur. For example, an electronic lock may require a user to press and release button A1, then A0, then A2 to unlock a door. A toll booth may raise a toll gate when the booth operator presses button A0, then keep the gate up as long as a car is detected by sensor A1 near the gate. For each system, the essential behavior involves not just input/output values but also the ~~order~~ of those input/output values over time.

Like most programming languages, C was not designed for time-ordered behavior. C uses a **sequential instructions computation model**, wherein statements (instructions) in a list are executed sequentially (one after another) until the list's end is reached. A sequential instructions model is good for capturing algorithms that transform given input data into output data, known as **data processing behavior**. However, the sequential instructions model is poorly-suited for capturing time-ordered behavior.

For example, consider a system on a carousel (merry-go-round) that increments B whenever the carousel rotates once, detected by a sensor that briefly pulses A0 for each rotation. The following RIMS code captures the system's behavior.

PARTICIPATION ACTIVITY

17.1.1: Pulse counting code for a carousel.



Animation content:

C program:

```
#include "RIMS.h"
void main()
{
    B = 0;
    while (1) {
        while (!A0);
        B = B + 1;
        while (A0);
    }
}
```

©zyBooks 07/21/23 23:56 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Animation captions:

1. A0 outputs 1 if the sensor near, otherwise A0 outputs 0.
2. Program waits in first loop while A0 is 0. Waits in the second loop while A0 is 1.
3. Carousel system increments B for each rotation.

The code's statements like `while (!A0);` may look unusual to a beginning embedded programmer. The statement loops as long as A0 is 0, and the immediate ; means no statements execute within the loop. Because C isn't designed to capture such time-oriented behavior, the code is slightly awkward, but understandable. However, the code becomes less understandable as more time-oriented behavior is introduced, such as having a button A1 that resets B, as shown below.

Figure 17.1.1: Extended carousel code: Becoming hard to understand due to sequential instructions model not made for capturing time-oriented behavior.

```
#include "RIMS.h"

void main()
{
    B = 0;
    while (1) {
        while (!A1 &&
!A0);
        if (A1) {
            B = 0; // Reset
        }
        else {
            B = B + 1;
            while (A0);
        }
    }
}
```

The code is becoming harder to understand. The text itself is simple, but how the code interacts with the inputs over time is not obvious, and requires plenty of mental execution to understand the system's behavior. With even more time-oriented behavior introduced, the code may become a spaghetti-like mess whose behavior is extremely hard to understand.

The lesson is this: Capturing time-ordered behavior directly into C's sequential instructions computation model is challenging. Instead, a computation model better suited for capturing time-ordered behavior is needed. State machines, introduced in another section, is one such model.

PARTICIPATION ACTIVITY

17.1.2: Time-ordered behavior.

©zyBooks 07/21/23 23:56 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Indicate whether each is more time-ordered behavior or data-processing behavior.

- 1) Raise a toll-gate, wait for a car to pass,
lower the toll gate.

Time-ordered



Data-processing

- 2) Find the maximum value in a set of 100 integers.

 Time-ordered Data-processing

- 3) Given two four-bit inputs, compute their sum and average as two four-bit outputs.

 Time-ordered Data-processing

©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



- 4) When a person is detected approaching the front of a door, automatically open the door until sensors in front of and behind the door no longer detect anybody.

 Time-ordered Data-processing

- 5) A wrong-way system has 10 sensors on a freeway offramp. If a car is driving the wrong way, the sensors will detect a car in the opposite order as normal. The system should flash a "Wrong way" sign and notify the police.

 Time-ordered Data-processing

17.2 State machines

©zyBooks 07/21/23 23:56 1692462

A **state machine** is a computation model intended for capturing time-ordered behavior. Numerous kinds of state machines exist. Common features of state machines are a set of inputs and outputs, a set of states with actions, a set of transitions with conditions, and an initial state. A drawing of a state machine is called a **state diagram**.

©zyBooks 07/21/23 23:56 1692462

COLORADOCSPB2270Summer2023

Consider a simple time-oriented system that turns on a light (by setting $B0 = 1$) if a user presses a button ($A0$ is 1 when pressed). The light stays on even after the button is released. Pressing a second button ($A1$ is 1) turns the light off. The following figure captures that behavior as a state machine.



Animation content:

Shown state machine has input A0 and A1 and output B0, and the states Unlit (with action B0 = 0) and Lit (with action B0 = 1).

©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea

COLORADO SPB2270Summer2023

The Unlit state has two transitions with conditions A0 and !A0. The !A0 condition transitions back to Unlit and A0 transitions to Lit.

The Lit state has two transitions with conditions A1 and !A1. The !A1 condition transitions back to Lit and A1 transitions to Unlit.

Animation captions:

1. The SM has has two inputs, A0 and A1, and one output, B0. The SM has two states Unlit and Lit. The Unlit state turns off the LED by setting B0 = 0. The Lit state turns on the LED by setting B0 = 1.
2. The SM implements a basic light toggle that turns on a light if a user presses the button connected to A0.
3. The light remains on even after the button to A0 is released. Pressing a second button, connected to A1, turns the light off.

The above animation shows a state machine with inputs A0 and A1 and output B0 (each 1 bit), and the states Unlit (with action B0 = 0) and Lit (with action B0 = 1). The state machine has four transitions with conditions !A0, A0, !A1, and A1, and the initial state is Unlit (denoted by the special "initial transition" arrow).

A system described by a state machine executes as follows. At any time, the system is "in" some state, called the **current state**. Upon starting, the transition to the initial state is taken and that state's actions are executed once. The following process then occurs, called a **tick** of the SM:

- A transition T leaving the current state and having a true condition is taken
- Transition T's target state has its actions executed once and becomes the current state

The ticking process repeats. Each tick takes a tiny but non-zero (perhaps nearly-infinitesimally small) amount of time, during which no event is assumed to occur. Ticks are assumed to occur at a much faster rate than input events, so no input events are missed. The following animation illustrates SM ticking.



Animation content:

Shown state machine has input A0 and A1 and output B0, and the states Unlit (with action $B0 = 0$) and Lit (with action $B0 = 1$).

The Unlit state has two transitions with conditions A0 and $\neg A0$. The $\neg A0$ condition transitions back to Unlit and A0 transitions to Lit.

The Lit state has two transitions with conditions A1 and $\neg A1$. The $\neg A1$ condition transitions back to Lit and A1 transitions to Unlit.

©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Animation captions:

1. The initial state is Unlit, so the LED is off.
2. Upon a tick, the SM remains in the Unlit state since a button has not been pressed. The $\neg A0$ transition evaluates to true. The LED remains off.
3. On the next tick, the SM transitions to the Lit state because the A0 transition evaluates to true. The Lit state sets $B0 = 1$, so the LED is turned on.

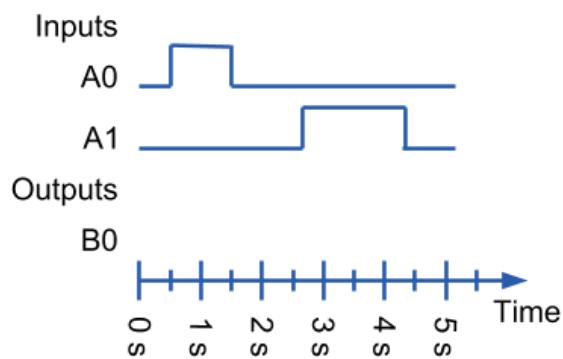
For the above example, upon startup the system takes the transition to state Unlit and executes $B0 = 0$ once. For subsequent ticks, if A0 is 0 then the system takes the transition back to state Unlit and executes $B0 = 0$ again. At some time, if A0 is 1 then the system takes the transition to state Lit and executes $B0 = 1$. The system stays in that state until A1 becomes 1, at which time the system takes the transition back to state Unlit.

PARTICIPATION ACTIVITY

17.2.3: Tracing execution of an SM.



Given the following timing diagram and the above light on/off SM, determine the value of B0 at the specified times.



©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

1) 0 s

- 1
 0

2) 1 s

- 1



0

3) 2 s

 1 0

©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

4) 3 s

 1 0

5) 4 s

 1 0

6) 5 s

 1 0

For the above input sequence, the light was on from about 0.5 seconds when A0's button was pressed, until about 2.5 seconds when A1's button was pressed.

If none of the current state's transitions has a true condition for a given tick, an implicit transition back to the state itself is taken (thus causing the state's actions to execute each such tick). Good practice, however, is to have an explicit transition point back to the same state with the proper condition, rather than relying on the implicit transition from a state to itself. The above system has an explicit transition with condition !A0 from Unlit back to Unlit, for example, making very clear what happens when in state Unlit and A0 is 0.

For a state machine to be precisely defined, transitions leaving a particular state should have **mutually exclusive** transition conditions, meaning only one condition could possibly be true at any time (otherwise, which of two transitions with true conditions should be taken? The state machine becomes **non-deterministic** in that case). For example, state Unlit transitions have conditions A0 and !A0, only one of which can possibly be true at any time.

©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea

A transition may be specified to have a condition of "true" or 1, meaning the transition should always be taken. That transition of course should be the only one leaving a particular state, else mutual exclusivity would not exist. A common shorthand notation omits the "true"; a transition with no condition is known to have a true condition.

A state may have multiple actions, such as `B0 = 1; B1 = 1;` or may have no actions at all. A state's actions execute once each time that a tick takes the state machine to that state, even if a

transition points back to the same state.

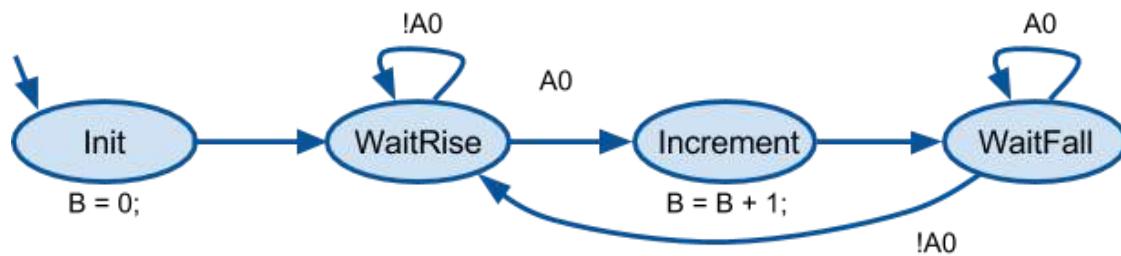
We will use a particular form of a state machine model, referred to in this material just as an **SM**, intended for creating C programs that support time-ordered behavior. An SM uses declared C variables rather than explicit inputs and outputs; the lone exception is the use of RIMS' implicitly-declared A and B input and output variables though. The SM's state actions consist of C statements, and the SM's transitions consist of C expressions. Variable values (such as $B0 = 1$) persist between ticks.

©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea

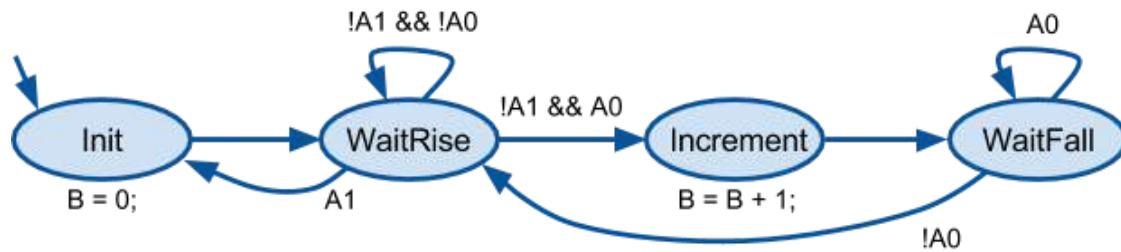
Using an SM, the earlier section's pulse counting system for a carousel can be captured as follows.³ Note how the SM more clearly defines the time-oriented behavior, versus the earlier section's awkward C code.

Figure 17.2.1: Pulse counting SM.



A reset behavior that returns the state machine to the initial state when A1 is pressed can be easily added, as shown below.

Figure 17.2.2: Pulse counting with reset SM.



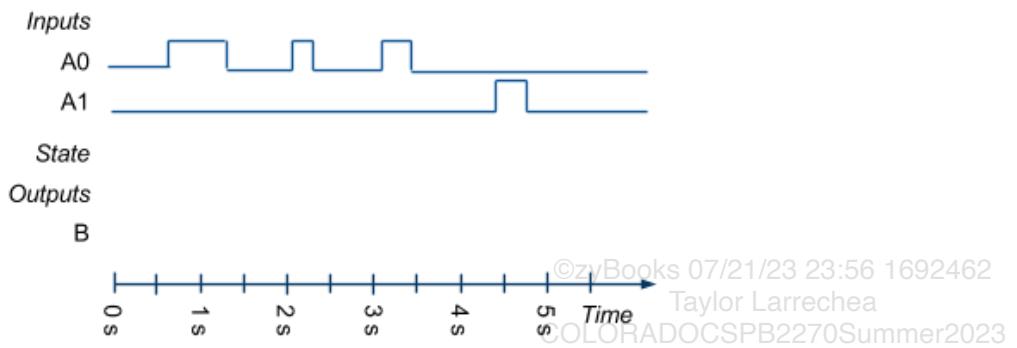
©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea

COLORADO CSPB2270Summer2023

The following timing diagram shows a sample sequence of inputs to the above SM.

Figure 17.2.3: Sample timing diagram for pulse counter with reset.

**PARTICIPATION ACTIVITY****17.2.4: Tracing the pulsing counting SM.**

Given the above SM input sequence, type the SM's current state at the specified times. At time 0 ms, the answer is: Init.

1) 0 s

 //**Check****Show answer**

2) 0.5 s

 //**Check****Show answer**

3) 1 s

 //**Check****Show answer**

4) 1.5 s

 //**Check****Show answer**

5) What is the integer value of B at time 4 s?

 //

©zyBooks 07/21/23 23:56 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Check**Show answer**

- 6) What is the integer value of B at time 5 s?

 //

©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Check**Show answer**

Try 17.2.1: Trace the current state.

Complete the above timing diagram by showing the current state and also the B0 signal value.

Notice that the SM model and the C code from an earlier section have the same behavior. However, the SM more explicitly captures the desired time-ordered behavior. This straightforwardness can be further seen by trying to extend the SM.

Try 17.2.2: Extend pulse counter to detect a threshold.

Extend the pulse counter SM to set B7 = 1 to indicate when the value of B reaches 99 (for the carousel system, such an indication may tell the ride operator to do a routine safety check). Note that 99 requires only the lower 7 bits of B, so B7 can be used for such indication. When 99 is reached, counting stops until a rising event on A1 resets the count.

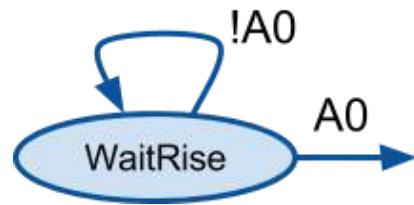
The following pattern is common in SMs for detecting a *rising* edge of a signal. The SM stays in the state while A0 is 0. When A0 becomes one, the transition is taken to another state. Detecting a falling edge is similar, with the condition swapped.

©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Figure 17.2.4: Pattern to detect rising edge of a bit signal.



©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

17.2.5: State machines.



1) An SM tick consists of:

- Executing the current state's actions and then transitioning to the next state.
- Transitioning to the next state and executing that state's actions an unknown # of times.
- Transitioning to the next state and executing that state's actions once.



2) How many SM ticks occur per second?

- 0
- 1
- Many
- Infinite



3) In the SM model, which is true about ticks and input events?

- Input events may occur faster than ticks.
- Two events may occur between ticks.
- An event may occur in the middle of a tick.
- Ticks occur faster than events.



©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Exploring further:

- [Wikipedia: Finite state machine](#)
- [Wikipedia: State diagram](#)

17.3 RIBS

©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea

The **RIBS** (Riverside-Irvine Builder of State machines) tool supports graphical state diagram capture of SMs.

PARTICIPATION
ACTIVITY

17.3.1: RIBS: Low and high example.

 Full screen



- Press "Simulate", observe SM executing. Press A0 to change to 1, note state and output changes. Press A0 again. Press "End simulation".
- Modify SM by adding action "B1 = 1;" to Lo, "B1 = 0;" to Hi. Simulate.
- Modify SM by inserting state All, sets B0 = 1, B1 = 1. Delete transition from Hi with !A0, insert transition with !A0 from Hi to All. Insert transition to stay in All while !A1, and another to go to Lo when A1. Simulate.

Simulate Pause Insert state Insert transition

LoHi x +

A0	0	<input type="checkbox"/>
A1	0	<input type="checkbox"/>
A2	0	<input type="checkbox"/>
A3	0	<input type="checkbox"/>
A4	0	<input type="checkbox"/>
A5	0	<input type="checkbox"/>
A6	0	<input type="checkbox"/>
A7	0	<input type="checkbox"/>
A = 0		

©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

<https://learn.zybooks.com/zybook/COLORADOCSPB2270Summer2023/chapter/17/print>

11/47

PARTICIPATION ACTIVITY

17.3.2: RIBS basics.



- 1) The SM's initial state is set by double-clicking the state.

©zyBooks 07/21/23 23:56 169246
Taylor Larrechea
COLORADOCSPB2270Summer2023

- True
 False

- 2) After inserting a state, actions can be added by typing in the "Actions" box on the far right.



- True
 False

- 3) A state or transition is deleted by dragging the item off-screen.



- True
 False

- 4) When an SM is executing (by pressing "Simulate"), the values of inputs A0, A1, A2, ..., cannot be changed.



- True
 False

PARTICIPATION ACTIVITY

17.3.3: RIBS with export/import.

Full screen



- SMs can be saved. Press "Export" and copy-paste the exported text. Modify the SM somehow. Then paste the text into the box and press "Import" -- the previously-exported SM is restored. (Users can save the exported text in a file or email for future use.)

©zyBooks 07/21/23 23:56 169246
Taylor Larrechea
COLORADOCSPB2270Summer2023

Simulate

Pause

Insert state

Insert transition

LoHi

A0 0 A1 0 A2 0 A3 0 A4 0 A5 0 A6 0 A7 0

A = 0

©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Export to RIMS

Export

Import

Lo hi



Paste design here.

PARTICIPATION ACTIVITY

17.3.4: RIBS with export.



©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



- 1) Pressing "Export" saves the current SM to a file.

 True False

17.4 Implementing an SM in C

Because microprocessors typically have C compilers but not SM compilers, implementing an SM in C is necessary. Using a standard method for implementing an SM to C enhances the readability and correctness of the resulting C code. The following illustrates such a method for the given SM named Latch (abbreviated as LA), which saves (or "latches") the value of A1 onto B0 whenever A0 is 1.

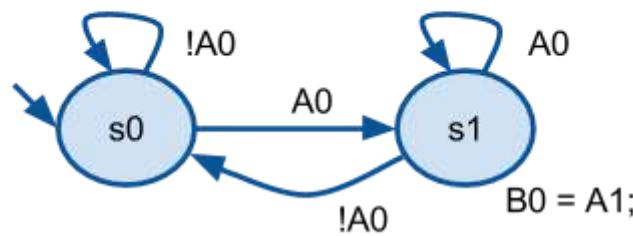
©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Figure 17.4.1: Method for implementing an SM in C.

SM name: Latch (abbrev: LA)



©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

```
#include "RIMS.h"

enum LA_States { LA_SMStart, LA_s0, LA_s1 }

LA_State;

void TickFct_Latch()
{
    switch(LA_State) { // Transitions
        case LA_SMStart: // Initial transition
            LA_State = LA_s0;
            break;

        case LA_s0:
            if (!A0) {
                LA_State = LA_s0;
            }
            else if (A0) {
                LA_State = LA_s1;
            }
            break;

        case LA_s1:
            if (!A0) {
                LA_State = LA_s0;
            }
            else if (A0) {
                LA_State = LA_s1;
            }
            break;

        default:
            LA_State = LA_SMStart;
            break;
    } // Transitions

    switch(LA_State) { // State actions
        case LA_s0:
            break;

        case LA_s1:
            B0 = A1;
            break;

        default:
            break;
    } // State actions
}

void main() {
    B = 0x00; // Initialize outputs
    LA_State = LA_SMStart; // Indicates initial call

    while(1) {
        TickFct_Latch();
    }
}
```

©zyBooks 07/21/23 23:56 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

While the code may seem imposing, the code follows a simple pattern:

- State variable declaration
- Tick function

- main loop that calls tick function

State variable declaration: The first line creates a new enum data type LA_States defined to have possible values of LA_SMStart, LA_s0 and LA_s1. That same code line declares global variable LA_State to be of type LA_States. **enum** is a C construct for defining a new data type, in contrast to built-in types like char or short, whose value can be one of an "enumerated" list of values. The programmer provides the enumeration list, as in { LA_SMStart, LA_s0, LA_s1 } above. In C, each item in the list becomes a new constant, with the first item having value 0, the second item having value 1, etc.

©zyBooks 07/21/23 23:56 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Tick function: The SM's **tick function** carries out one tick of the SM. This SM's tick function is named TickFct_Latch(). For the current state LA_State, the tick function's *first switch statement* takes the appropriate transition to a new current state; if LA_State is LA_SMStart, the transition is to the SM's initial state. The *second switch statement* then executes the actions for the new current state.

main loop that calls tick function: main() first initializes outputs; good practice for any embedded program, whether implementing an SM or not, is to start the main function by initializing all outputs. main() then sets the current state to a value of LA_SMStart. main() then enters the normal infinite "while (1)" loop, which just repeatedly calls the function TickFct_Latch() to repeatedly tick the Latch SM.

PARTICIPATION ACTIVITY

17.4.1: Follow an SMs execution in C.

 Full screen



Run the above code in RIMS. Press "Break" and then repeatedly press "Step" (setting A0 accordingly), observing how the code executes each tick of the state machine.

©zyBooks 07/21/23 23:56 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

A0 0

A1 0

A2 0

A3 0

A4 0

©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

A5 0

A6 0

A7 0

A = 0

```
1 #include "RIMS.h"
2 int main() {
3     while(1) {
4         B0 = A0 && A1;
5     }
6     return 0;
7 }
```

©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

B0 0
B1 0
B2 0
B3 0
B4 0
B5 0
B6 0
B7 0
B = 0

©zyBooks 07/21/23 23:56 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Compile Run Stop Continue Break Step

Simulation speed: Normal ▾

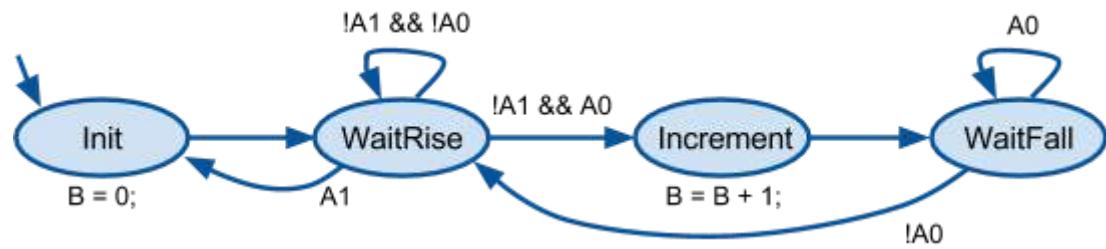
Generate timing diagram

0.00 s

The transition switch statement's default case should never actually execute, but good practice is to always include a default case for a switch statement, in case something bad happens. For a state machine, the default case should be included for safety if the state variable ever somehow gets corrupted. The default case commonly has a transition to the initial state, causing the SM to start over rather than getting stuck.

Capturing behavior as an SM and then converting to C using the above method typically results in more code than capturing behavior directly in C. However, *more code does not mean worse code.* The C code generated from an SM may be more likely to be correct, may be more easily extensible and maintainable, and has other major benefits that will be seen later.

Figure 17.4.2: Carousel SM in C.



©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

©zyBooks 07/21/23 23:56 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

©zyBooks 07/21/23 23:56 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

```

#include "RIMS.h"

enum CR_States { CR_SMStart, CR_Init, CR_WaitRise, CR_Increment, CR_WaitFall }
CR_State;

void TickFct_Carousel()
{
    switch(CR_State) { // Transitions
        case CR_SMStart: // Initial transition
            CR_State = CR_Init;
            break;

        case CR_Init:
            CR_State = CR_WaitRise;
            break;

        case CR_WaitRise:
            if (A1) {
                CR_State = CR_Init;
            }
            else if (!A1 && !A0) {
                CR_State = CR_WaitRise;
            }
            else if (!A1 && A0) {
                CR_State = CR_Increment;
            }
            break;

        case CR_Increment:
            CR_State = CR_WaitFall;
            break;

        case CR_WaitFall:
            if (!A0) {
                CR_State = CR_WaitRise;
            }
            else if (A0) {
                CR_State = CR_WaitFall;
            }
            break;

        default:
            CR_State = CR_SMStart;
            break;
    } // Transitions

    switch(CR_State) { // State actions
        case CR_Init:
            B = 0;
            break;
        case CR_WaitRise:
            break;
        case CR_Increment:
            B = B + 1;
            break;
        case CR_WaitFall:
            break;
        default:
            break;
    } // State actions
}

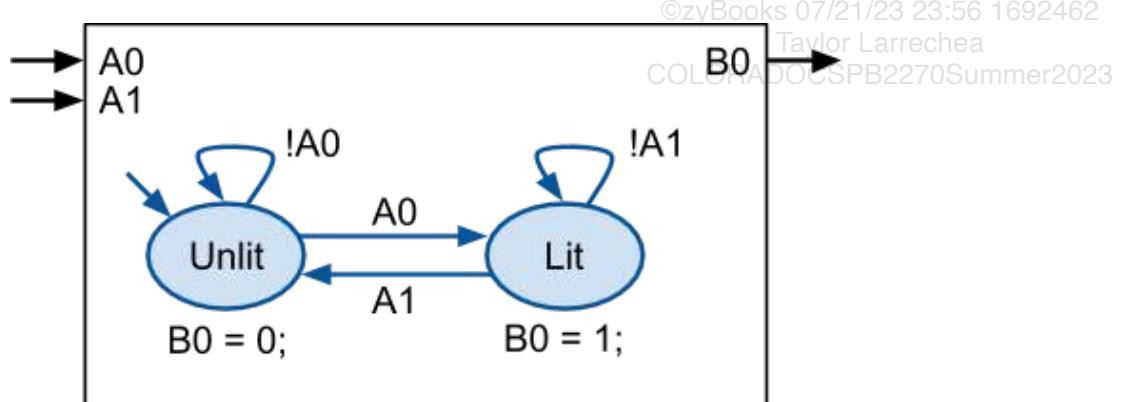
void main() {
    B = 0x00; // Initialize outputs
    CR_State = CR_SMStart; // Indicates initial call
    while(1) {
        TickFct_Carousel();
    }
}

```

©zyBooks 07/21/23 23:56 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

For the following question sets, consider the SM below.

Figure 17.4.3: Light on/off system.



PARTICIPATION ACTIVITY

17.4.2: Method for implementing an SM in C.



Strictly adhere to the above SM to C implementation method, including naming conventions. Use LT_States as the enumerated type name, and TickFct_LightToggle as the tick function name.

- 1) Provide the enumerated type definition following the above SM to C method.

//

Check

Show answer



- 2) main()'s while(1) loop will consist of what one statement?

//

Check

Show answer



©zyBooks 07/21/23 23:56 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

17.4.3: C code for SM.





- 1) The tick function's first switch statement will execute the current state's actions.
- True
 False
- 2) The first switch statement's first case will include: case LT_SMStart: LT_State = LT_Unlit;
- True
 False
- 3) The first switch statement's second case will be for the transitions going to state Unlit.
- True
 False
- 4) The tick function's second switch statement will include: case (LT_Unlit):
if (!A0) { B0 = 1;...}
- True
 False
- 5) The SM to C method uses an if-else statement for multiple transitions, but could have just used multiple if statements.
- True
 False
- 6) If a state has no actions, the state should be omitted from the second case statement.
- True
 False
- 7) The break statements could be removed from the switch statements without changing behavior, but should be included for clarity.

©zyBooks 07/21/23 23:56 169246
Taylor Larrechea
COLORADOCSPB2270Summer2023



©zyBooks 07/21/23 23:56 169246
Taylor Larrechea
COLORADOCSPB2270Summer2023



- True
 - False
- 8) A transition from a state back to that same state can be omitted from the first switch statement without changing the SM's behavior.
- True
 - False



©zyBooks 07/21/23 23:56 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

**PARTICIPATION
ACTIVITY**

17.4.4: SM to C for the light toggle system.

Full screen



Strictly following the above method for implementing an SM in C, implement the above light toggle SM in C.

©zyBooks 07/21/23 23:56 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

A0 0

A1 0

A2 0

A3 0

A4 0

©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

A5 0

A6 0

A7 0

A = 0

```
1 #include "RIMS.h"
2 int main() {
3     while(1) {
4         B0 = A0 && A1;
5     }
6     return 0;
7 }
```

©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

B0 0

B1 0

B2 0

B3 0

B4 0

B5 0

B6 0

B7 0

B = 0

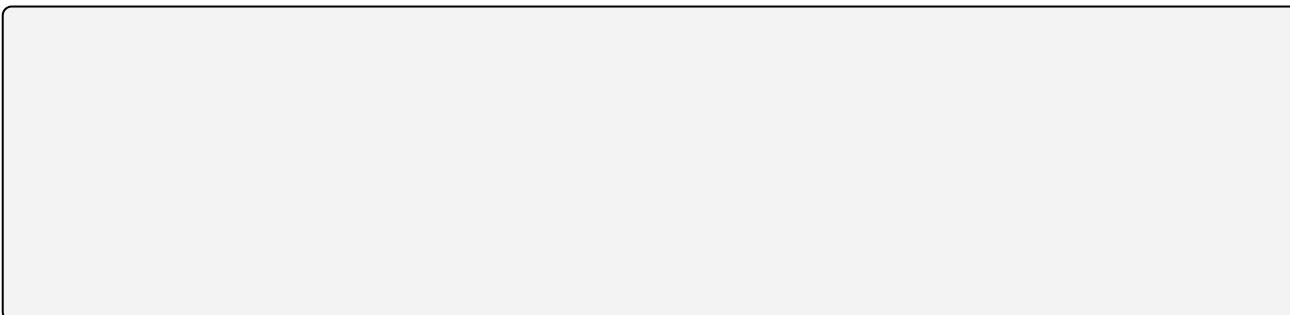
©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

CompileRunStopContinueBreakStepSimulation speed: Generate timing diagram

0.00 s

**CHALLENGE ACTIVITY**

17.4.1: Complete SM for given behavior.

Full screen



489394.3384924.qx3zqy7

Start

©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

A0 0

A1 0

A2 0

A3 0

A4 0

©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

A5 0

A6 0

A7 0

A = 0

1

©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

B0 0

B1 0

B2 0

B3 0

B4 0

B5 0

B6 0

B7 0

B = 0

©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

CompileRunStop

0.00 s

1

2

3

4

5

CheckNext

17.5 Variables, statements, and conditions in SMs

©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea

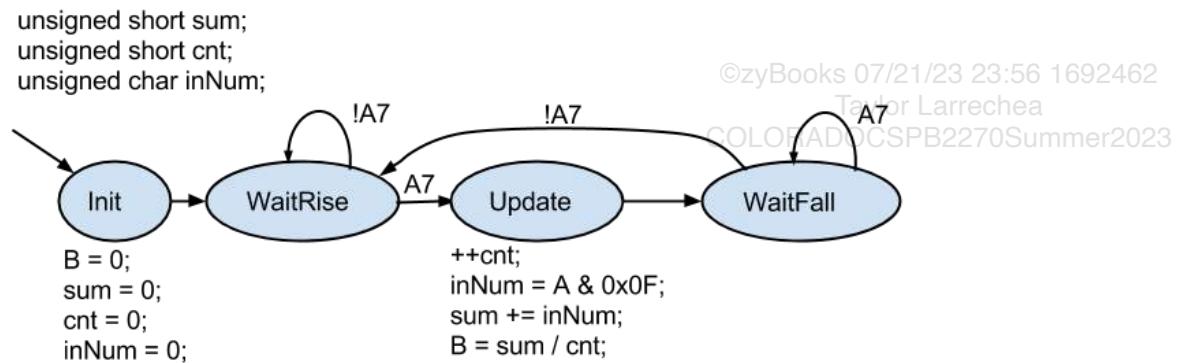
COLORADOCSPB2270Summer2023

Variables

An SM can have variables declared at the SM scope (i.e., not within a state), which can be accessed by all actions and conditions of the SM. For example, the following SM uses several variables to output

the average of 4-bit numbers that appear on A3..A0 when A7 rises.

Figure 17.5.1: SM using variables to compute average.



©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

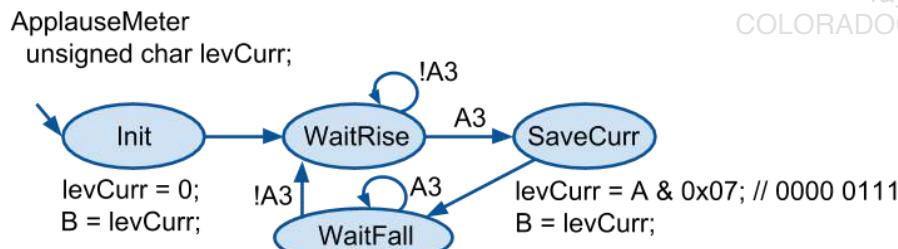
Variable sum maintains the sum of all 4-bit numbers seen so far, defined as a short to reduce likelihood of overflow. Variable cnt counts the number of times A7 has risen. Variable inNum stores A3..A0, that variable used just to improve code readability. (Note: A reset for the above system, perhaps using A6, is omitted for brevity).

In RIMS, the input/output variables like A0 or B can be thought of as having been declared as `unsigned char` variables.

While the variables could be initialized when declared (e.g., `unsigned short sum=0;`), good practice is to create a state named Init to carry out initializations of variables and also outputs. In this way, not only are all initializations in one place, but the system can be re-initialized merely by transitioning back to the Init state.

Example 17.5.1: Applause meter.

Consider an applause-meter system intended for a game show. A sound sensor measures sound on a scale of 0 to 7 (0 means quiet, 7 means loud), outputting a three-bit binary number, connected to RIM's A2-A0. A button connected to A3 can be pressed by the game show host to save (when A3 rises) the current sound level, which will then be displayed on B. The system's behavior can be captured as an SM with a variable, as shown:



©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

When converting to C, the SM variables can be implemented as global variables above the SM's tick function.

PARTICIPATION ACTIVITY

17.5.1: SM variables.



- 1) A programmer can declare variables within each SM state.

True
 False

©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

- 2) An SM variable maintains its value across SM ticks.

True
 False

- 3) Failing to write an SM variable in a particular state causes that variable to become 0.

True
 False



- 4) When implementing an SM in C, the SM's variables should be declared in the main() function.

True
 False



Try 17.5.1: System to count instances driving without seatbelt.

Design a system for an automobile that counts the number of times the car was put into drive (A0 is 1) while the driver's seatbelt was not fastened (A1 is 0). Once the car starts driving, do no further counting until the car is taken out of drive (A0 is 0). While the car is not in drive, a mechanic can view the count by holding a button (A2 is 1) causing the count to appear on B. B is normally 0.

While the car is not in drive, a mechanic can view the count by holding a button (A2 is 1) causing the count to appear on B. B is normally 0.
©zyBooks 07/21/23 23:56 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Statements

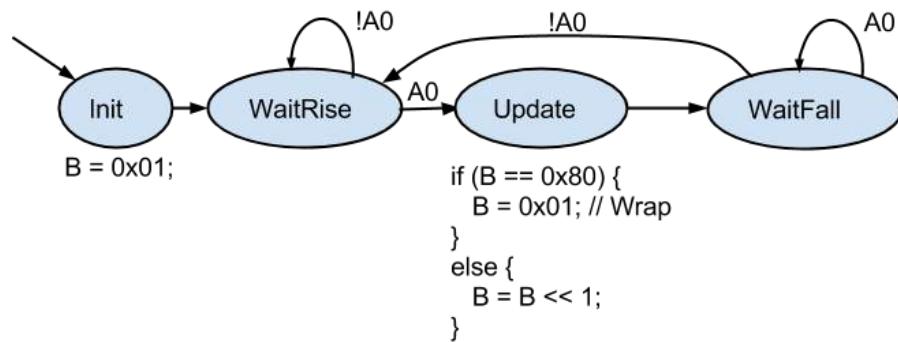
The statements that can appear in actions can include more than just assignment statements. Any C statements can appear, such as if-else statements, loops, and function calls. However, for this material's purposes, good practice is to ensure that actions don't wait on an external input value, such as `while (!A0) {};`. Waiting behavior should be captured as states and transitions so that all time-ordered behavior is visible at the transition level. Also, a state with actions that wait could cause the SM to violate the basic assumption that SM ticks occur faster than events so that no events are missed.

©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea
COLORADOCSPB2270Summer2023

The following example changes RIMS' output LED pattern from 00000001 to 00000010 to 00000100, etc., each time A0 rises. When 10000000 is reached, the pattern wraps back to 00000001.

Figure 17.5.2: SM using if-else statement to output a shifting pattern.



Try 17.5.2: Extend the applause meter SM.

Extend the above applause meter SM so that if A4 becomes 1 while in the WaitRise state, the system outputs the maximum value seen so far. When A4 is returned to 0, the system outputs the most-recently-saved value as before. Use a variable for the most-recently-saved value and another for the max value, and an if-else statement.

Try 17.5.3: SM with loop.

©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea
COLORADOCSPB2270Summer2023

Create an SM that waits for A0 to rise, upon which the SM counts the number of 1s on A1..A7 and outputs the count on B. That count stays on B until A0 rises again. Use the GetBit function from an earlier section, and a for loop, as the actions of a state that counts the 1s on A1..A7.



1) A state's actions may include a for loop.

- True
 False

©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



2) A state's actions may include a function call.

- True
 False



3) The statement `if (A0) { ... }` should not appear in a state's actions.

- True
 False



4) The statement `while (A0) { ... }` should not appear in a state's actions.

- True
 False

Conditions

Conditions in an SM should be C expressions. The following rule is important:

- *Exactly one—no more, no fewer—of a state's exiting transitions should have a condition that evaluates to true at a given time.* In this way, the next state for each tick is precisely and clearly defined.

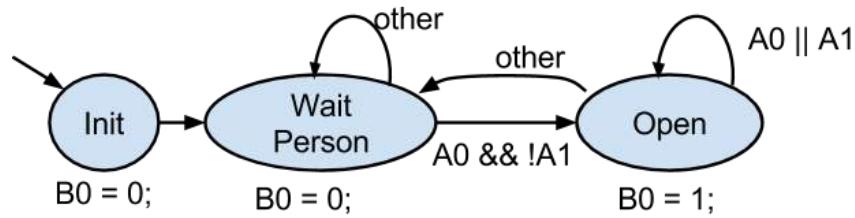
A common error is to create transitions leaving a state whose conditions are not mutually exclusive, like one transition with condition A0 and another with condition A1, both of which could be true simultaneously. Another common error is to create transitions such that sometimes no transition has a true condition, like one transition with condition A0 and another with condition !A0 && A1, which fail to cover the situation !A0 && !A1. Technically, neither situation is actually an error. In the first case, the SM becomes non-deterministic, because the model does not define which of two true transitions will be taken. In the second case, the SM will implicitly take a transition back to the same state, but explicit transitions are preferable for clarity.

For convenience, the SM's in this material use a condition named **other** to indicate the transition that should be taken if none of the state's normal transition conditions are true. For example, the following system opens a swinging door when a person approaches the front (A0 is 1) and nobody is directly

behind the door (A_1 is 0). Note that the transition that remains in the `WaitPerson` state is simply "other" rather than being $!(A_0 \&& !A_1)$, which clutters the SM and detracts the reader's attention away from the more important transition $A_0 \&& !A_1$ from `WaitPerson` to `Open`. Once opening the door, the system keeps the door open as long as the person is detected near the door as indicated by $A_0 || A_1$, again using "other" for the opposite condition.

Figure 17.5.3: Door opener system using condition other.

©zyBooks 07/21/23 23:56 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



When implementing an SM in C, the "other" transition may be implemented as a last else branch (with no expression) in the state's transitions if-else code.

PARTICIPATION ACTIVITY

17.5.3: SM conditions.



Transition conditions are expressions, *not* statements, so should *not* end with a semicolon.

For each, write the most direct answer.

Ex: For "A1 and A0 are true", write

A1 && A0

without parentheses, semicolons, and without **==**.

- 1) Write the condition: either A_1 or A_0 is true.



//

Check

Show answer

- 2) Write the condition that detects that A is greater than or equal to 99.

©zyBooks 07/21/23 23:56 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



//

Check

Show answer



- 3) Write the condition that detects that A2 A1 A0 is 010. Use individual bit variables A2, A1, A0.

Check**Show answer**

- 4) A designer intended to have one transition taken if A0 is 1; otherwise, a second transition is taken if A1 is 0 and a third taken if A1 is 1. The designer wrote the conditions **A0**, **!A1**, and **A1**, which are not mutually exclusive. Fix the second condition.

Check**Show answer**

- 5) A designer intended to have one transition taken if exactly one of A1 or A0 is 1, and a second transition taken if both are 0s. The designer wrote the conditions as **(A1 && !A0) || (!A1 && A0)** and as **!A1 && !A0**. A third transition is missing; write its condition.

Check**Show answer**

- 6) A designer has two transitions leaving a state. One transition's condition is **(A1 || A2 || A3)**. The second transition's condition is "other". What expression is equivalent to "other"?

Check**Show answer**

©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023





7) A designer has three transitions leaving a state. One transition's condition is $(\neg A_1 \And \neg A_0)$. The second condition is $(A_1 \And A_0)$. The third transition's condition is "other". What expression is equivalent to "other"?

Hint: Use parentheses, and don't simplify. Start with: $\neg(\neg A_1$

©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

 Check

Show answer

17.6 Mealy actions

The earlier state machine model associates actions with states only, known as a **Moore-type state machine**. A **Mealy-type state machine** allows actions on transitions too. A Mealy-type SM can make some behaviors easier to capture.

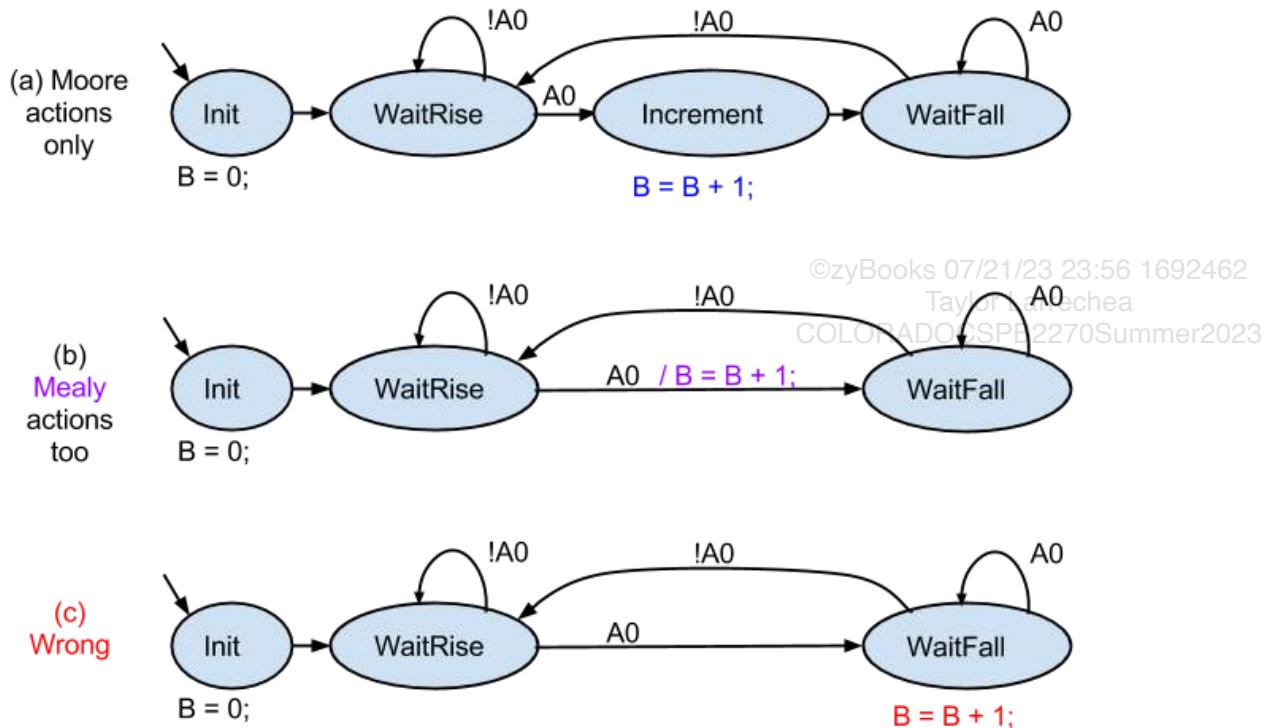
For example, the following figure parts (a) and (b) show SMs that increment variable cnt once for each rising A0. The Mealy SM increments on the transition that detected the rise, and thus avoids the need for a separate state. Note that Mealy actions are shown graphically following a transition's condition and a /.

Figure 17.6.1: Mealy SMs can reduce states and more intuitively represent some behavior.

©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea

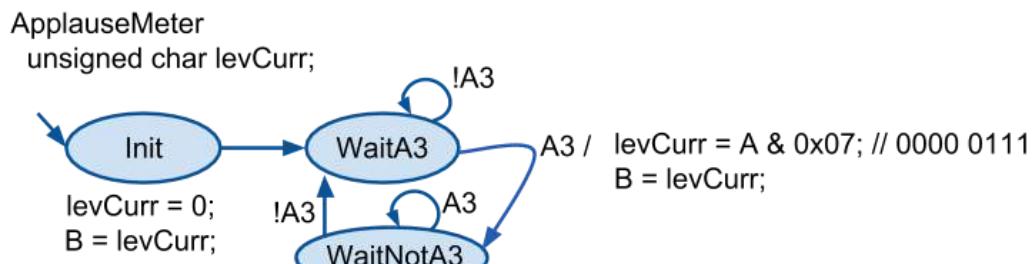
COLORADOCSPB2270Summer2023



Note that the SM in (c) is wrong, incrementing B repeatedly while A0 is 1. Removing the A0 transition from WaitFall would still be wrong because the SM model would have an implicit transition back to the state if no other transition is true.

The following implements an applause meter system (from an earlier section), using a Mealy action on the transition that detects a rising A3, thus preventing having an additional state.

Figure 17.6.2: Applause meter using a Mealy action.



©zyBooks 07/21/23 23:56 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

When translating to C, a transition's actions appear in the transition switch statement, in the appropriate if-else branch.

Try 17.6.1: MealySMToggle.

Capture a toggle light system using an SM. A button connects to A0. B0 connects to a light, initially off. Pressing the button (rising A0) changes the light from off to on. Another press changes the light from on to off. And so on. First try capturing with a Moore SM. Then try with an SM having Mealy actions, and notice that fewer states are required.

PARTICIPATION ACTIVITY

17.6.1: Mealy actions.

©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



1) A Mealy action occurs at which point during an SM tick?

- While checking whether a transition's condition is true or false.
- While taking a transition to the next state.
- Upon entering a new state.
- Before an SM's tick.



2) Can a Mealy action include an if-else statement?

- No, only one statement is allowed.
- No, only assignments statements are allowed.
- Yes, but the if and else must both assign the same variable.
- Yes.



3) Integer variable X is 0. A transition with action $X = X + 1$ points to a state with action $X = X + 2$. If that transition is taken during an SM tick, what is X after the tick?

- 0
- 1
- 2
- 3



©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

17.7 How to capture behavior as an SM

Capturing behavior as an SM is an art. The following 3-step process may help. Consider an emergency exit door that sounds an alarm when armed (switch A0 is 1) and then opened (door A1 is 1).

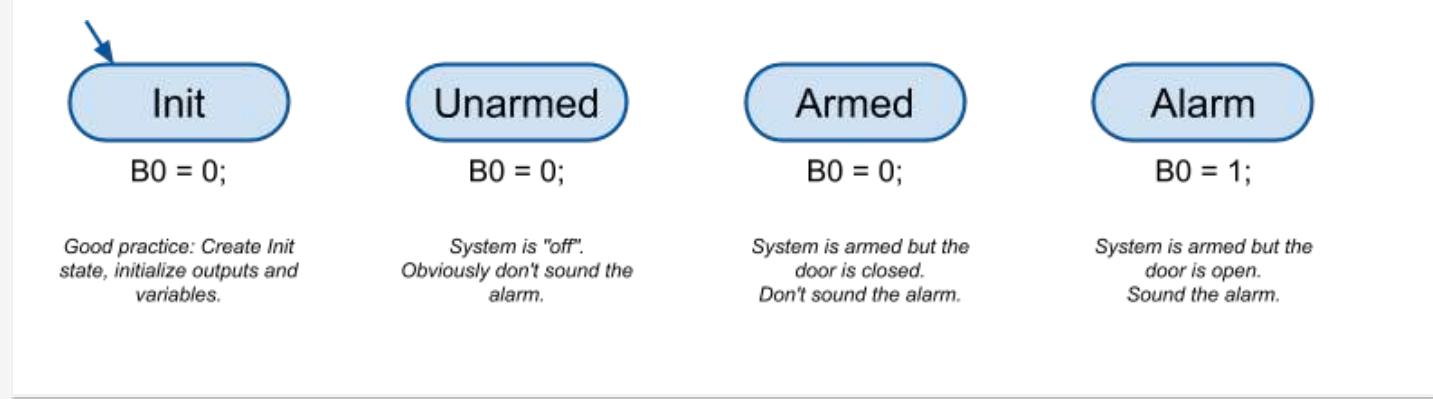
Step 1: A first step is to list the system's basic states, adding actions if known:

©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea

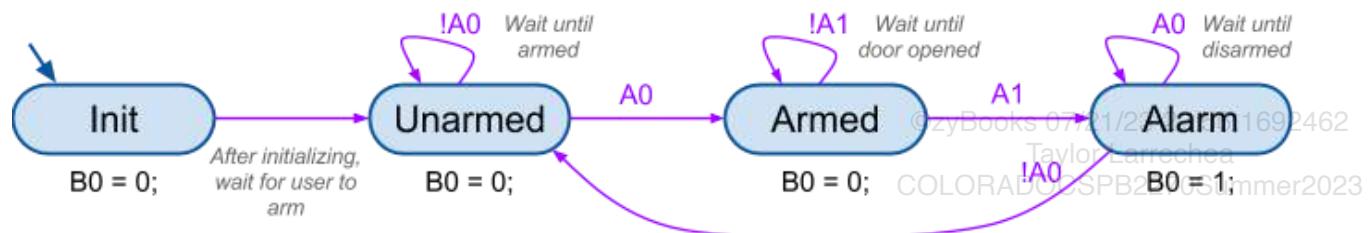
COLORADOCSPB2270Summer2023

Figure 17.7.1: Emergency exit door system: Basic states.



Step 2: A second step of the process is to add transitions to each state to achieve the desired behavior. Clearly the first state to enter after initialization is the Unarmed state, so we add a conditionless (true) transition from Init to Unarmed. We add a transition to stay in Unarmed while A0 is 0, and another transition to go to Armed when A0 is 1. We add a transition to stay in Armed while the door is closed (A1 is 0), and another transition to go to Alarm if the door is opened. Finally, we add a transition to stay in Alarm while the system is still armed (A0 is 1), and when disarmed (A0 is 0) we could go to either Init or Unarmed; Unarmed seems reasonable so we add a transition to there.

Figure 17.7.2: Emergency exit door system: Adding transitions (first attempt).



Step 3: The third step is to mentally check the behavior of the captured SM. This step may result in adding more transitions, or even more states. Minimally, for each state, we should check that exactly

one transition's condition will be true, modifying the conditions or adding transitions if necessary. Even more importantly, we should also determine whether the SM's behavior is as desired, by mentally executing the SM and thinking what input sequences might occur. In doing so for the above SM, we consider the possibility that a user might want to disarm the system while armed, and notice that is not possible in the SM. Thus, we may add transitions, such as !A0 from Armed to Unarmed (requiring that the other two transitions be modified, otherwise the state's outgoing conditions would not be mutually exclusive).

©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

When mentally executing, one might focus on a particular state, and then for each input not explicitly on the transitions one might ask: "Does that input's value matter for this state's behavior?" For example, the above system has two inputs, A0 and A1. For state Unarmed, does input A1's value matter? It does not for that state. For state Armed, does input A0 matter? It does, so we need to add transitions. For state Alarm, does A1's value matter? It does not; once the door has been opened, the alarm continues sounding whether the door stays open or closed.

Capturing time-ordered behavior is hard. Good designers will spend much time mentally executing their SM and thinking of possible input sequences that need to be addressed. Refining an SM many times is commonplace.

PARTICIPATION ACTIVITY

17.7.1: Capturing behavior as an SM.



- 1) The above process suggests creating one state at a time, creating that state's actions and all the states outgoing transitions, before moving on to create another state.

- True
 False



- 2) Step 3 for the emergency door example determined that a transition with condition A0 is needed from Armed to Unarmed.

- True
 False



- 3) Upon adding a transition !A0 from Armed to Unarmed, the transition A1 from Armed should be changed to A1 && A0.

- True
 False

©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



Try 17.7.1: Exit only doorway.

A doorway is for exit only. Sensors A0, A1, A2 detect a person passing through. A proper exit causes A2A1A0 to be 000, then 100, then 010, then 001, then 000. Any other sequence causes a buzzer to sound ($B_0=1$) until 000.

Design an SM using RIBS that captures the doorway behavior, and simulate using RIMS.

©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Try 17.7.2: Automatic door.

An automatic door at a store has a sensor in front (A0) and behind (A1). The door opens ($B_0=1$) when a person approaches from the front, and stays open as long as a person is detected in front or behind. If the door is closed and a person approaches from the behind, the door does not open. If the door is closed and a person approaches from the front but a person is also detected behind, the door does not open, to prevent hitting the person that is behind.

Design an SM using RIBS that captures the automatic door behavior, and simulate using RIMS.

Try 17.7.3: Dimmer light system.

A dimmer light system has increase (A0) and decrease (A1) buttons. B sets the light intensity, 0 is off, 255 is the maximum intensity, and:

- Pressing both buttons does nothing.
- Pressing both buttons immediately turns the light off.

Design an SM using RIBS that captures the dimmer light behavior, and simulate using RIMS.

©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Try 17.7.4: Amusement ride.

An amusement park ride has sensor mats on the left (A0) and right (A1) of a ride car. A ride operator starts the ride by pressing a button (A7); each unique press toggles the ride from stopped to started ($B_0=1$) and vice-versa. If anyone leaves the ride car and steps on a sensor mat, the ride stops and an alarm sounds ($B_1=1$). The alarm stops sounding when the person

gets off the sensor mat. The only way for the ride to restart is for the operator to press the button again. The ride never starts if someone is on the sensor mat.

Design an SM using RIBS that captures the amusement ride behavior, and simulate using RIMS.

©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

17.8 Testing an SM

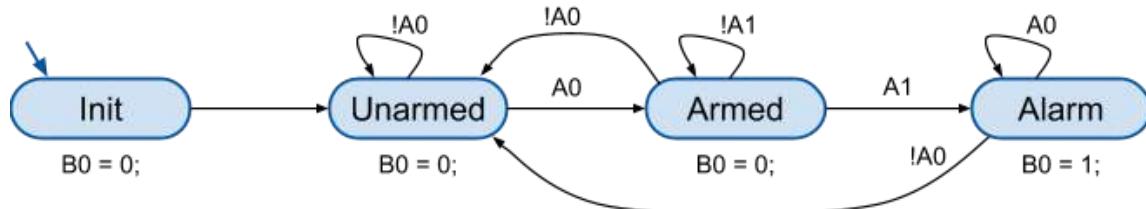
Testing time-ordered behavior requires generating a good set of input scenarios, where a **scenario** is a sequence of inputs that should cause a particular sequence of state. Such testing is in contrast to merely generating a good set of input combinations for a system lacking internal states, each input combination known as a **test vector**.

A testing process for time-ordered behavior may consist of:

1. Decide what scenarios to test, including normal cases as well as border cases.
2. Devise a sequence of test vectors to test each scenario.

Consider the following SM for an emergency exit alarm system.

Figure 17.8.1: Emergency exit door system to be tested.



The following lists some scenarios, and corresponding test vectors.

Table 17.8.1: A few test scenarios for the emergency door system.

©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Scenario. Starting state in ()		Expected final state	Expected output (B0)	Test vectors (A1A0)
1	(Init): Alarm should not sound.	Unarmed	0	00
2	(Unarmed): Arm system: Alarm should not sound.	Armed	0	01

3	(Armed): Open door. <i>Alarm should sound.</i>	Alarm	1	11
4	(Alarm): Unarm: <i>Alarm should stop.</i>	Unarmed	0	10
5	(Unarmed): Arm, open door, close door. <i>Alarm should continue to sound.</i>	Alarm	1 ©zyBooks 07/21/23 23:56 1692462 Taylor Larrachea COLORADOCSPB2270Summer2023	00, 01, 11, 01
6	(Alarm): Unarm, arm, unarm: <i>Alarm should not sound.</i>	Unarmed	0	00, 01, 00

Each row indicates the assumed starting state (same as previous row's final state), describes the system's input scenario and the expected system behavior, lists the expected final state and output, and finally lists the test vectors that will generate the scenario. For example, the scenario 2 starts in state Unarmed, then arms to the system, which should cause a change to state Armed and output B0=1. Arming the system is achieved by A1A0 being 01, which is the test vector. Scenario 5 carries out a longer sequence, requiring several test vectors.

Scenario 6 arms and then unarms the system. Testing with the given test vectors will yield the correct final output of 0, but the final state will be Armed rather than Unarmed. Thus, testing uncovers a problem, namely that Armed is missing a transition with condition !A0 going back to Unarmed .

As with testing systems with combinational behavior, testing a system with time-ordered behavior should involve normal cases and border cases. Border cases specifically test unusual situations, like all inputs changing from 0s to 1s simultaneously, or an input changing back and forth from 0 to 1 repeatedly.

When testing an SM, test vectors should ensure that *each state and each transition is executed at least once*. Furthermore, if a state's action code has branches, then test vectors should also ensure that every statement is executed at least once. Even more ideally, *every path* through the SM would also be tested, but achieving complete path coverage is hard because huge numbers of paths may exist.

In testing terminology, **black-box testing** refers to checking for correct outputs only, as in checking the value of B0 in the above example. In **white-box testing**, internal values of the system are also checked, such as the current state. One can see the advantage of white-box testing in the above example, where the output was correct but the state was not. Further black-box testing might discover the above problem too, but white-box is more likely to detect problems. Of course, white-box testing is harder because a mechanism is necessary to access internal values.

Designing good test vectors can take much effort, both to formulate the scenarios, and to properly create the test vectors. Testing is as important as capturing the SM. Good practice is to plan to spend nearly as much time for testing a system as for designing a system. New programmers rarely follow this practice, believing testing is just a "sanity check" step at the end of design.

**PARTICIPATION
ACTIVITY**

17.8.1: Creating test vectors for the emergency door system.



Match the test vectors with the desired test scenario for the above emergency door system. Assume each set of test vectors is applied immediately after starting the system, so the starting state is always Unarmed. Each vector is for A1A0, so 01 means A1 is 0 and A0 is 1.

If unable to drag and drop, refresh the page.

©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

01, 10**01, 00, 10****01, 00, 01, 00****01, 11, 01, 11**

Arm the system, open the door, close the door, open the door.

Arm the system, disarm, arm, disarm.

Arm the system, disarm the system, open the door.

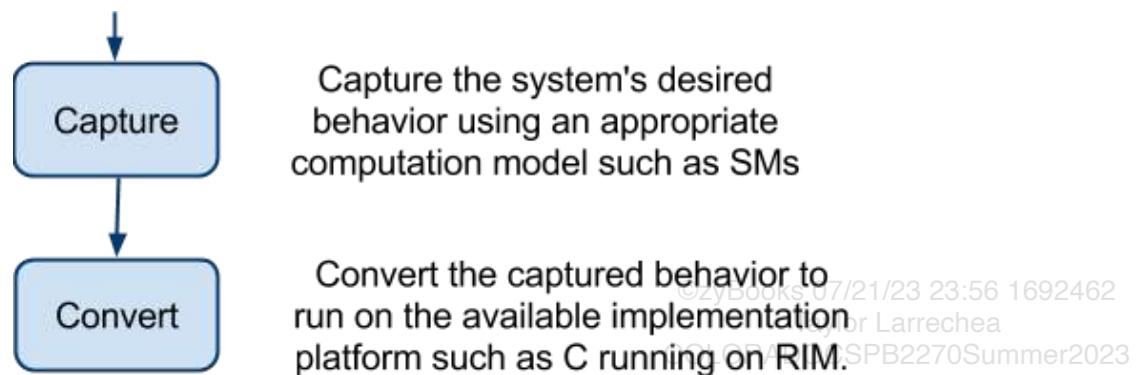
Arm the system, then simultaneously open the door and disarm the system.

Reset

17.9 Capture/convert process

The above sections described a two-step process that is common in disciplined embedded programming. The first step is to **capture** the desired behavior using a computation model appropriate for the desired system behavior, such as SMs. The second step is to **convert** that captured behavior into an implementation, such as C that will run on a microprocessor. The conversion is typically very structured and automatable. The capture/convert process will be used in subsequent chapters for more complex behavior and can result in code that is more likely to be correct, that is more maintainable, and that has many other benefits compared to behavior that is captured directly in C's sequential instruction model. *The capture/convert design process is perhaps one of the most important concepts in disciplined programming of embedded systems.*

Figure 17.9.1: Capture, convert.



©zyBooks 07/21/23 23:56 1692462

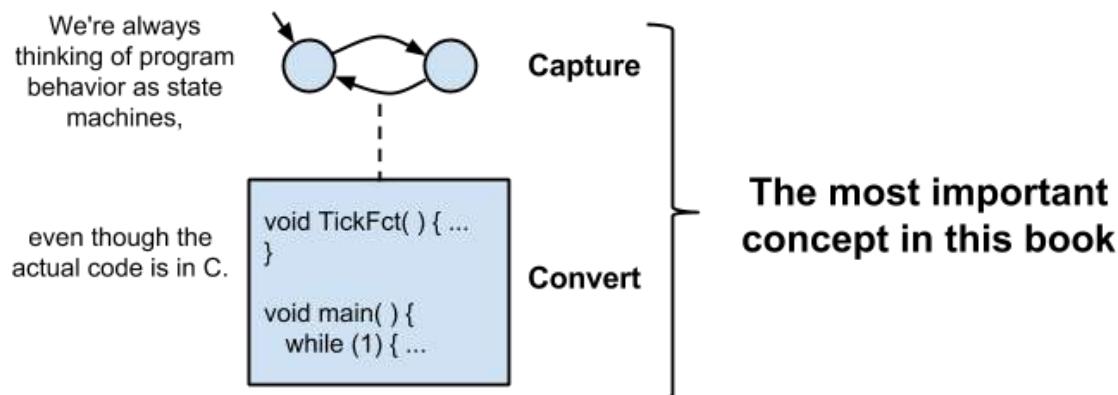
Taylor Larrechea

COLORADOCSPB2270Summer2023

Even in the absence of a tool like RIBS, programmers can (and should) capture time-ordered behavior as an SM, typically drawing the SM on paper first, and then converting to C. All modifications are done by changing the SM (capture), and then updating the C code (convert). An experienced programmer can work with SMs in C without always having to see a state diagram, and can see or draw a state diagram from C code.

The concept of thinking of program behavior as state machines, even though the actual code is in C, is the most important concept in this book.

Figure 17.9.2: Thinking in state machines.



Try 17.9.1: Reverse engineering C code to an SM.

©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

For the below C code, draw the corresponding SM state diagram.

```
#include "RIMS.h"

unsigned char x;

enum EX_States { EX_SMStart, EX_S0, EX_S2, EX_S1 }
EX_State;

void TickFct_Example() {
    switch(EX_State) { // Transitions
        case EX_SMStart:
            EX_State = EX_S0;
            break;
        case EX_S0:
            if (1) {
                EX_State = EX_S1;
            }
            break;
        case EX_S2:
            if (A3) {
                EX_State = EX_S2;
            }
            else if (!A3) {
                EX_State = EX_S1;
            }
            break;
        case EX_S1:
            if (!A3) {
                EX_State = EX_S1;
            }
            else if (A3) {
                EX_State = EX_S2;
                x = A & 0x07;
                B = x;
            }
            break;
        default:
            EX_State = EX_SMStart;
            break;
    }

    switch(EX_State) { // State actions
        case EX_S0:
            x = 0;
            B = x;
            break;
        case EX_S2:
            break;
        case EX_S1:
            break;
        default:
            break;
    }
}

void main() {
    EX_State = EX_SMStart; // Initial state
    B = 0; // Init outputs
    while(1) {
        TickFct_Example();
    }
}
```

©zyBooks 07/21/23 23:56 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Note that the RIBS tool does not run a C compiler on the C code in actions/conditions/declarations, but rather just generates a new C program that contains that code. Any syntax errors will only be

determined upon running a C compiler on that program, requiring a RIBS user to correlate the error message to the SM code.

Try 17.9.2: C syntax error in RIBS.

For any working example in RIBS having an action in a state, introduce a C syntax error by removing the semicolon after an action. Save, generate C, then press "RIMS/Simulation". Note the error message that is generated, and strive to correlate that with the RIBS SM. Fix the error and this time introduce an error by adding a semicolon after a transition condition, and try running again.

The SM model in this chapter is a basic state machine model specifically intended to aid the capture of time-ordered behavior and for conversion to C. Many other state machine models exist, such as

UML state machines. Some state machine models have a more formal mathematical basis, but translation to C is more cumbersome and the code harder to maintain. Another common category of computation model involves dataflow models, which are well suited to digital signal processing applications but are beyond our scope.

PARTICIPATION ACTIVITY

17.9.1: Capture/convert process.



- 1) "Capture" refers to describing desired system behavior in the most appropriate computation model.

- True
- False



- 2) "Convert" refers to describing desired system behavior directly in C.

- True
- False



- 3) A system has two 4-bit inputs, and continually sets a 5-bit output to their average. The system is best captured as an SM, then converted to C.

- True
- False

©zyBooks 07/21/23 23:56 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



- 4) Because an SM can be converted to C, then C directly supports the SM



computation model.

- True
- False

©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Exploring further:

- [Wikipedia: UML state machine](#)
- [statecharts.org](#)
- [Wikipedia: Kahn process networks](#)

©zyBooks 07/21/23 23:56 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

18.1 Huffman compression

Basic compression idea

Given data represented as some quantity of bits, **compression** transforms the data to use fewer bits. Compressed data uses less storage and can be communicated faster than uncompressed data.

The basic idea of compression is to encode frequently-occurring items using fewer bits. Ex: Uncompressed ASCII characters use 8 bits each, but compression uses fewer than 8 bits for more frequently occurring characters.

PARTICIPATION ACTIVITY

18.1.1: The basic idea of compression is to use fewer bits for frequent items (and more bits for less-frequent items).



Animation content:

undefined

Animation captions:

1. The text "AAA Go" as ASCII would use $6 * 8 = 48$ bits. Such data is uncompressed.
2. Compression uses a dictionary of codes specific to the data. Frequent items get shorter codes. Here, A (which is most frequent) is 0, space 10, G 110, and o 111.
3. Thus, "AAA Go" is compressed as 0 0 0 10 110 111. The compressed data uses only eleven bits, much fewer than the 48 bits uncompressed.

PARTICIPATION ACTIVITY

18.1.2: Basic compression.



Given the following dictionary:

00000000: 00

11111111: 01

00000010: 10

00000011: 110

00000100: 111

©zyBooks 07/21/23 23:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 1) Compress the following: 00000000
00000000 11111111 00000100



Check**Show answer**

- 2) Compress the following: 00000011
00000010

//

Check**Show answer**

©zyBooks 07/21/23 23:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

- 3) Decompress the following: 00 01 00

//

Check**Show answer**

- 4) Is any code in the dictionary a prefix of another code? Type yes or no.

//

Check**Show answer**

- 5) Decompress the following, in which the spaces that were inserted above for reading convenience are absent:
0011000.

//

Check**Show answer**

Building a character frequency table

Prior to compression, a character frequency table must be built for any input string. Such a table contains each distinct character from the input string and each character's number of occurrences.

Programming languages commonly provide a dictionary or map object to store the character frequency table.

PARTICIPATION ACTIVITY

18.1.3: Building a character frequency table.

Animation content:

Static figure: A code block and a character frequency table.

Begin pseudocode:

```
BuildCharacterFrequencyTable(inputString) {  
    table = new Dictionary()  
    for (i = 0; i < inputString.length; i++) {  
        currentCharacter = inputString[i]  
        if (table has key for currentCharacter) {  
            table[currentCharacter] = table[currentCharacter] + 1  
        }  
        else {  
            table[currentCharacter] = 1  
        }  
    }  
    return table  
}
```

©zyBooks 07/21/23 23:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

BuildCharacterFrequencyTable("APPLES AND BANANAS")

End pseudocode.

Step 1: A new dictionary is created for the character frequency table and iteration through the input string's characters begins.

The code BuildCharacterFrequencyTable("APPLES AND BANANAS") is highlighted. Then, the code BuildCharacterFrequencyTable(inputString) { is highlighted, followed by the code table = new Dictionary(). The text table: (empty) appears below the code block. The code for (i = 0; i < inputString.length; i++) { is highlighted. The text i: 0 appears below the code block. The code currentCharacter = inputString[i] is highlighted. The text currentCharacter: A appears under the text i: 0.

Step 2: The current character, A, is not in the dictionary. So A is added to the dictionary with a frequency of 1.

The code if (table has key for currentCharacter) { is highlighted. The text not in table appears next to the text currentCharacter: A. The code table[currentCharacter] = 1 is highlighted. The text (empty) disappears after the word table under the code block. A new entry, A 1, appears in the table.

©zyBooks 07/21/23 23:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Step 3: The next character, P, is also not in the dictionary and is added with a frequency of 1.

The code for (i = 0; i < inputString.length; i++) { is highlighted, followed by the code currentCharacter = inputString[i]. The letter A is replaced with a P after the text currentCharacter. The code if (table has key for currentCharacter) { is highlighted. The text not in table appears next to the text currentCharacter: P. The code table[currentCharacter] = 1 is highlighted. A new entry, P 1, appears in the table.

Step 4: The next character, P, is already in the table, so the existing frequency is incremented from 1 to 2.

The code `currentCharacter = inputString[i]` is highlighted, and the P after the text `currentCharacter` is highlighted. The code `if (table has key for currentCharacter) {` is highlighted. The text in table appears next to the text `currentCharacter: P`. The code `table[currentCharacter] = table[currentCharacter] + 1` is highlighted. The P entry in the table is updated from 1 to 2.

Step 5: For remaining characters, first occurrences set the frequency to 1 and subsequent occurrences increment the existing frequency.

©zyBooks 07/21/23 23:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

The entire for loop is highlighted. The frequency table is filled in: A 5, P 2, L 1, E 1, S 2, (space) 2, N 3, D 1, B 1.

Animation captions:

1. A new dictionary is created for the character frequency table and iteration through the input string's characters begins.
2. The current character, A, is not in the dictionary. So A is added to the dictionary with a frequency of 1.
3. The next character, P, is also not in the dictionary and is added with a frequency of 1.
4. The next character, P, is already in the table, so the existing frequency is incremented from 1 to 2.
5. For remaining characters, first occurrences set the frequency to 1 and subsequent occurrences increment the existing frequency.

PARTICIPATION ACTIVITY

18.1.4: Character frequency counts.



Given the text "seems he fled", indicate the frequency counts.

1) s

- 1
- 2
- 3



2) e

- 2
- 5
- 6



©zyBooks 07/21/23 23:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

3) Each m, h, f, l, and d

- 1
- 2



- 3
- 4) (space)
 - 1
 - 2
 - 3



©zyBooks 07/21/23 23:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Huffman coding

Huffman coding is a common compression technique that assigns fewer bits to frequent items, using a binary tree.

PARTICIPATION ACTIVITY

18.1.5: A binary tree can be used to determine the Huffman coding.



Animation captions:

1. Huffman coding first determines the frequencies of each item. Here, a occurs 4 times, b 3, c 2, and d 1. (Total is 10).
2. Each item is a "leaf node" in a tree. The pair of nodes yielding the lowest sum is found, and merged into a new node formed with that sum. Here, c and d yield $2 + 1 = 3$.
3. The merging continues. The lowest sum is b's 3 plus the new node's 3, yielding 6. (Note that c and d are no longer eligible nodes). The merging ends when only 1 node exists.
4. Each leaf node's encoding is obtained by traversing from the top node to the leaf. Each left branch appends a 0, and each right branch appends a 1, to the code.

PARTICIPATION ACTIVITY

18.1.6: Huffman coding example: Merging nodes.



A 100-character text has these character frequencies:

- A: 50
C: 40
B: 4
D: 3
E: 3

©zyBooks 07/21/23 23:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 1) What is the first merge?



- D and E: 6
- B and D: 7
- B and D and E: 10

2) What is the second merge?

- B and D: 7
- DE and B: 10
- C and A: 90



3) What is the third merge?

- DEB and C: 40
- DEB and C: 50



©zyBooks 07/21/23 23:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

4) What is the fourth merge?

- None
- DEBC and A: 100



5) What is the fifth merge?

- None
- DEBCA and F



6) What is the code for A?

- 0
- 1



7) What is the code for C?

- 1
- 01
- 10



8) What is the code for B?

- 001
- 110



9) What is the code for D?

- 1110
- 1111



©zyBooks 07/21/23 23:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

10) What is the code for E?

- 1110
- 1111





11) 5 unique characters (A, B, C, D, E) can each be uniquely encoded in 3 bits (like 000, 001, 010, 011, and 100). With such a fixed-length code, how many bits are needed for the 100-character text?

- 100
- 300

©zyBooks 07/21/23 23:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

12) For the Huffman code determined in the above questions, the number of bits per character is A: 1, C: 2, B: 3, D: 4, and E: 4. Recalling the frequencies in the instructions, how many bits are needed for the 100-character text?

- 14
- 166
- 300



Note: For Huffman encoded data, the dictionary must be included along with the compressed data, to enable decompression. That dictionary adds to the total bits used. However, typically only large data files get compressed, so the dictionary is usually a tiny fraction of the total size.

Building a Huffman tree

The data members in a Huffman tree node depend on the node type.

- Leaf nodes have two data members: a character from the input and an integer frequency for that character.
- Internal nodes have left and right child nodes, along with an integer frequency value that represents the sum of the left and right child frequencies.

A Huffman tree can be built from a character frequency table.

PARTICIPATION
ACTIVITY

18.1.7: Building a Huffman tree.

©zyBooks 07/21/23 23:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



Animation content:

undefined

Animation captions:

1. The character frequency table is built for the input string, BANANAS.
2. A leaf node is built for each table entry and enqueue in a priority queue. Lower frequencies have higher priority.
3. Leaf nodes for S and B are removed from the queue. A parent is built with the sum of the frequencies and is enqueue into the priority queue.
4. The two nodes with frequency 2 are dequeued and the parent with frequency $2 + 2 = 4$ is built.
5. The remaining 2 nodes are dequeued and given a parent.
6. When the priority queue has 1 node remaining, that node is the tree's root. The root is dequeued and returned.

©zyBooks 07/21/23 23:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

18.1.8: HuffmanBuildTree function.



Assume `HuffmanBuildTree("zyBooks")` is called.

- 1) The character frequency table has _____ entries.
 - 5
 - 6
 - 7
- 2) The leaf node at the back of the priority queue, `nodes`, before the while loop begins is _____.
 - z | 1
 - B | 1
 - o | 2
- 3) The parent node for nodes B and k has a frequency of _____.
 - 1
 - 2
 - 4



©zyBooks 07/21/23 23:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Implementing with 1 node structure

Implementations commonly use the same node structure for leaf and internal nodes, instead of two distinct structures. Each node has a frequency, character, and 2 child

pointers. The child pointers are set to null for leaves and the character is set to 0 for internal nodes.

Getting Huffman codes

©zyBooks 07/21/23 23:58 1692462

Taylor Larrechea
COLORADOCSPB2270Summer2023

Huffman codes for each character are built from a Huffman tree. Each character corresponds to a leaf node. The Huffman code for a character is built by tracing a path from the root to that character's leaf node, appending 0 when branching left or 1 when branching right.

PARTICIPATION
ACTIVITY

18.1.9: Getting Huffman codes.



Animation content:

undefined

Animation captions:

1. Huffman codes are built from a Huffman tree. Left branches add a 0 to the code, right branches add a 1.
2. HuffmanGetCodes starts at the root node with an empty prefix.
3. A recursive call is made on the root's left child. The node is a leaf and A's code is set to the current prefix, 0.
4. The first recursive call completes. The next recursive call is made for node 4 and a prefix of "1".
5. Node 4 and node 2 are not leaves, so additional recursive calls are made.
6. B's code is set to 100.
7. The remaining recursive calls set codes for S and N.
8. Each distinct character has a code. Characters B and S have lower frequencies than A and N and thus have longer codes.

PARTICIPATION
ACTIVITY

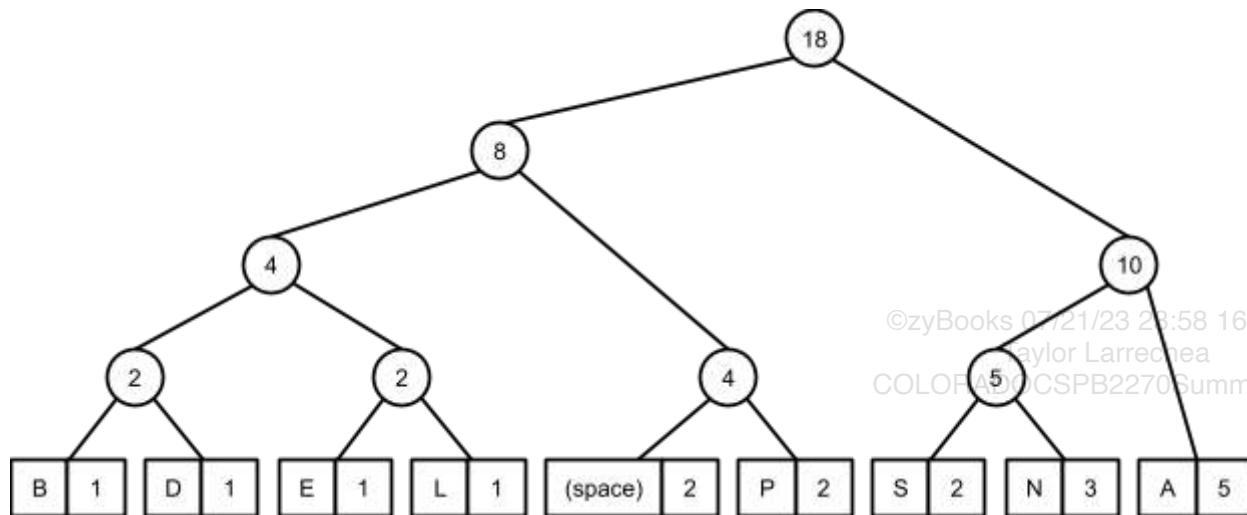
18.1.10: Huffman codes.



Below is the Huffman tree for "APPLES AND BANANAS"

©zyBooks 07/21/23 23:58 1692462

Taylor Larrechea
COLORADOCSPB2270Summer2023



©zyBooks 07/21/23 23:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

1) What is the Huffman code for A?

- 10
- 11
- 101



2) What is the Huffman code for P?

- 01
- 0000
- 011



3) What is the length of the longest
Huffman code?

- 3
- 4
- 5



Compressing data

To compress an input string, the Huffman codes are first obtained for each character. Then each character of the input string is processed, and corresponding bit codes are concatenated to produce the compressed result.

©zyBooks 07/21/23 23:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Figure 18.1.1: HuffmanCompress function.

```
HuffmanCompress(inputString) {  
    // Build the Huffman tree  
    root = HuffmanBuildTree(inputString)  
  
    // Get the compression codes from the tree  
    codes = HuffmanGetCodes(root, "", new  
Dictionary())  
  
    // Build the compressed result  
    result = ""  
    for c in inputString {  
        result += codes[c]  
    }  
    return result  
}
```

©zyBooks 07/21/23 23:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION
ACTIVITY

18.1.11: Compressing data.



Match each compression call to the result. Spaces are added between character codes for clarity, but would not exist in the actual compressed data.

If unable to drag and drop, refresh the page.

HuffmanCompress("BANANAS")

HuffmanCompress("aabbbac")

HuffmanCompress("zyBooks")

100 0 11 0 11 0 101

00 101 010 11 11 011 100

11 11 0 0 0 11 10

Reset

©zyBooks 07/21/23 23:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Decompressing Huffman coded data

To decompress Huffman code data, one can use a Huffman tree and trace the branches for each bit, starting at the root. When the final node of the branch is reached, the result has been found. The process continues until the entire item is decompressed.



Animation captions:

1. The Huffman code is decompressed by first starting at the root. The branches are followed for each bit.
2. When the final node of the branch is reached, the result has been found.
3. Once the final node is reached decoding restarts at the root node.
4. The process continues until the entire item is decompressed.

©zyBooks 07/21/23 23:58 1692462
Taylor Larrechea

COLORADOCSPB2270Summer2023

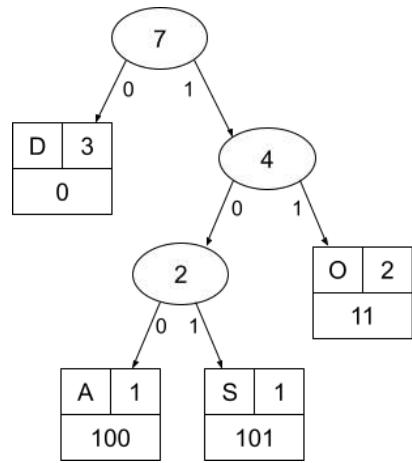
Figure 18.1.2: HuffmanDecompress function.

```
HuffmanDecompress(compressedString, treeRoot) {  
    node = treeRoot  
    result = ""  
    for (bit in compressedString) {  
        // Go to left or right child based on bit value  
        if (bit == 0)  
            node = node->left  
        else  
            node = node->right  
  
        // If the node is a leaf, add the character to  
        // the  
        // decompressed result and go back to the root  
        node  
        if (node is a leaf) {  
            result += node->character  
            node = treeRoot  
        }  
    }  
    return result  
}
```



Use the tree below to decompress 0111101000101.

©zyBooks 07/21/23 23:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



©zyBooks 07/21/23 23:58 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

- 1) What is the first decoded character?

Check

Show answer



- 2) What is the second decoded character?

Check

Show answer



- 3) 11 yields the third character O.
 0 yields the fourth character D.



What is the next decoded character?

Check

Show answer

- 4) What is the decoded text?



Check

Show answer

©zyBooks 07/21/23 23:58 1692462
 Taylor Larrechea
 COLORADOCSPB2270Summer2023

CHALLENGE ACTIVITY

18.1.1: Huffman compression.



Start

Complete the character frequency table for the string "BEES AND TREES".

Character	Frequency
(space)	2
A	Ex: 4
B	
D	
E	
N	
R	
S	
T	

©zyBooks 07/21/23 23:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

1

2

3

Check**Next**

18.2 Heuristics



This section has been set as optional by your instructor.

Heuristics

©zyBooks 07/21/23 23:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

In practice, solving a problem in the optimal or most accurate way may require more computational resources than are available or feasible. Algorithms implemented for such problems often use a **heuristic**: A technique that willingly accepts a non-optimal or less accurate solution in order to improve execution speed.

PARTICIPATION ACTIVITY

18.2.1: Introduction to the knapsack problem.

**Animation content:**

undefined

Animation captions:

©zyBooks 07/21/23 23:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

1. A knapsack is a container for items, much like a backpack or bag. Suppose a particular knapsack can carry at most 30 pounds worth of items.
2. Each item has a weight and value. The goal is to put items in the knapsack such that the weight ≤ 30 pounds and the value is maximized.
3. Taking a 20 pound item with an 8 pound item is an option, worth \$142.
4. If more than 1 of each item can be taken, 2 of item 1 and 1 of item 4 provide a better option, worth \$145.
5. Trying all combinations will give an optimal answer, but is time consuming. A heuristic algorithm may choose a simpler, but non-optimal approach.

PARTICIPATION ACTIVITY

18.2.2: Heuristics.



- 1) A heuristic is a way of producing an optimal solution to a problem.



- True
 False

- 2) A heuristic technique used for numerical computation may sacrifice accuracy to gain speed.



- True
 False

PARTICIPATION ACTIVITY

18.2.3: The knapsack problem.

©zyBooks 07/21/23 23:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



Refer to the example in the animation above.

- 1) Which of the following options provides the best value?



- 5 6-pound items

- 2 6-pound items and 1 18-pound item
 - 3 8-pound items and 1 6-pound item.
- 2) The optimal solution has a value of \$162 and has one of each item: 6-pound, 8-pound, and 18-pound.
- True
 - False
- 3) Which approach would guarantee finding an optimal solution?
- Taking the largest item that fits in the knapsack repeatedly until no more items will fit.
 - Taking the smallest item repeatedly until no more items will fit in the knapsack.
 - Trying all combinations of items and picking the one with maximum value.

©zyBooks 07/21/23 23:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Heuristic optimization

A **heuristic algorithm** is an algorithm that quickly determines a near optimal or approximate solution. Such an algorithm can be designed to solve the **0-1 knapsack problem**: The knapsack problem with the quantity of each item limited to 1.

A heuristic algorithm to solve the 0-1 knapsack problem can choose to always take the most valuable item that fits in the knapsack's remaining space. Such an algorithm uses the heuristic of choosing the highest value item, without considering the impact on the remaining choices. While the algorithm's simple choices are aimed at optimizing the total value, the final result may not be optimal.

PARTICIPATION
ACTIVITY

18.2.4: Non-optimal, heuristic algorithm to solve the 0-1 knapsack.

Animation content:

undefined

Animation captions:

1. The item list is sorted and the most valuable item is put into the knapsack first.
2. No remaining items will fit in the knapsack.
3. The resulting value of \$95 is inferior to taking the 12 and 8 pound items, collectively worth \$102.
4. The heuristic algorithm sacrifices optimality for efficiency and simplicity.

©zyBooks 07/21/23 23:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

18.2.5: Heuristic algorithm and the 0-1 knapsack problem.



1) Which is not commonly sacrificed by a heuristic algorithm?

- speed
- optimality
- accuracy



2) What restriction does the 0-1 knapsack problem have, in comparison with the regular knapsack problem?

- The knapsack's weight limit cannot be exceeded.
- At most 1 of each item can be taken.
- The value of each item must be less than the item's weight.



3) Under what circumstance would the Knapsack01 function not put the most valuable item into the knapsack?

- The item list contains only 1 item.
- The weight of the most valuable item is greater than the knapsack's limit.

©zyBooks 07/21/23 23:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Self-adjusting heuristic

A **self-adjusting heuristic** is an algorithm that modifies a data structure based on how that data structure is used. Ex: Many self-adjusting data structures, such as red-black trees and AVL trees, use a self-adjusting heuristic to keep the tree balanced. Tree balancing organizes data to allow for faster access.

Ex: A self-adjusting heuristic can be used to speed up searches for frequently-searched-for list items by moving a list item to the front of the list when that item is searched for. This heuristic is self-adjusting because the list items are adjusted when a search is performed.

©zyBooks 07/21/23 23:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

PARTICIPATION
ACTIVITY

18.2.6: Move-to-front self-adjusting heuristic.



Animation content:

undefined

Animation captions:

1. 42 is at the end of a list with 8 items. A linear search for 42 compares against 8 items.
2. The move-to-front heuristic moves 42 to the front after the search.
3. Another search for 42 now only requires 1 comparison. 42 is left at the front of the list.
4. A search for 64 compares against 8 items and moves 64 to the front of the list.
5. 42 is no longer at the list's front, but a search for 42 need only compare against 2 items.

PARTICIPATION
ACTIVITY

18.2.7: Move-to-front self-adjusting heuristic.



Suppose a move-to-front heuristic is used on a list that starts as (56, 11, 92, 81, 68, 44).

- 1) A first search for 81 compares against how many list items?

- 0
- 3
- 4

- 2) A subsequent search for 81 compares against how many list items?

- 1
- 2
- 4

- 3) Which scenario results in faster searches?

©zyBooks 07/21/23 23:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



- Back-to-back searches for the same key.
- Every search is for a key different than the previous search.

©zyBooks 07/21/23 23:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

18.3 Greedy algorithms



This section has been set as optional by your instructor.

Greedy algorithm

A **greedy algorithm** is an algorithm that, when presented with a list of options, chooses the option that is optimal at that point in time. The choice of option does not consider additional subsequent options, and may or may not lead to an optimal solution.

PARTICIPATION
ACTIVITY

18.3.1: MakeChange greedy algorithm.



Animation content:

undefined

Animation captions:

1. The change making algorithm uses quarters, dimes, nickels, and pennies to make change equaling the specified amount.
2. The algorithm chooses quarters as the optimal coins, as long as the remaining amount is ≥ 25 .
3. Dimes offer the next largest amount per coin, and are chosen while the amount is ≥ 10 .
4. Nickels are chosen next. The algorithm is greedy because the largest coin \leq the amount is always chosen.
5. Adding one penny makes 91 cents.
6. This greedy algorithm is optimal and minimizes the total number of coins, although not all greedy algorithms are optimal.

PARTICIPATION ACTIVITY

18.3.2: Greedy algorithms.



1) If the MakeChange function were to make change for 101, what would be the result?

- 101 pennies
- 4 quarters and 1 penny
- 3 quarters, 2 dimes, 1 nickel, and 1 penny

©zyBooks 07/21/23 23:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



2) A greedy algorithm is attempting to minimize costs and has a choice between two items with equivalent functionality: the first costing \$5 and the second costing \$7. Which will be chosen?

- The \$5 item
- The \$7 item
- The algorithm needs more information to choose



3) A greedy algorithm always finds an optimal solution.

- True
- False

Fractional knapsack problem

The **fractional knapsack problem** is the knapsack problem with the potential to take each item a fractional number of times, provided the fraction is in the range [0.0, 1.0]. Ex: A 4 pound, \$10 item could be taken 0.5 times to fill a knapsack with a 2 pound weight limit. The resulting knapsack would be worth \$5.

©zyBooks 07/21/23 23:58 1692462

COLORADOCSPB2270Summer2023

While a greedy solution to the 0-1 knapsack problem is not necessarily optimal, a greedy solution to the fractional knapsack problem is optimal. First, items are sorted in descending order based on the value-to-weight ratio. Next, one of each item is taken from the item list, in order, until taking 1 of the next item would exceed the weight limit. Then a fraction of the next item in the list is taken to fill the remaining weight.

Figure 18.3.1: FractionalKnapsack algorithm.

```
FractionalKnapsack(knapsack, itemList, itemListSize) {  
    Sort itemList descending by item's (value / weight)  
    ratio  
    remaining = knapsack->maximumWeight  
    for each item in itemList {  
        if (item->weight <= remaining) {  
            Put item in knapsack  
            remaining = remaining - item->weight  
        }  
        else if (remaining > 0) {  
            fraction = remaining / item->weight  
            Put (fraction * item) in knapsack  
            break  
        }  
    }  
}
```

zyBooks 07/21/23 23:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY**18.3.3: Fractional knapsack problem.**

Suppose the following items are available: 40 pounds worth \$80, 12 pounds worth \$18, and 8 pounds worth \$8.

- 1) Which item has the highest value-to-weight ratio?

- 40 pounds worth \$80
- 12 pounds worth \$18
- 8 pounds worth \$8



- 2) What would FractionalKnapsack put in a 20-pound knapsack?

- One 12-pound item and one 8-pound item
- One 40-pound item
- Half of a 40-pound item



- 3) What would FractionalKnapsack put in a 48-pound knapsack?

- One 40-pound item and one 8-pound item
- One 40-pound item and 2/3 of a 12-pound item

©zyBooks 07/21/23 23:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



Activity selection problem

The **activity selection problem** is a problem where 1 or more activities are available, each with a start and finish time, and the goal is to build the largest possible set of activities without time conflicts. Ex: When on vacation, various activities such as museum tours or mountain hikes may be available. Since vacation time is limited, the desire is often to engage in the maximum possible number of activities per day.

©zyBooks 07/21/23 23:58 1692462

Taylor Larrechea
COLORADOCSPB2270Summer2023

A greedy algorithm provides the optimal solution to the activity selection problem. First, an empty set of chosen activities is allocated. Activities are then sorted in ascending order by finish time. The first activity in the sorted list is marked as the current activity and added to the set of chosen activities. The algorithm then iterates through all activities after the first, looking for a next activity that starts after the current activity ends. When such a next activity is found, the next activity is added to the set of chosen activities, and the next activity is reassigned as the current. After iterating through all activities, the chosen set of activities contains the maximum possible number of non-conflicting activities from the activities list.

PARTICIPATION ACTIVITY

18.3.4: Activity selection problem algorithm.



Animation content:

undefined

Animation captions:

1. Activities are first sorted in ascending order by finish time. The set of chosen activities initially has the activity that finishes first.
2. The morning mountain hike does not start after the history museum tour finishes and is not added to the chosen set of activities.
3. The boat tour is the first activity to start after the history museum tour finishes, and is the "greedy" choice.
4. Hang gliding and the fireworks show are chosen as 2 additional activities.
5. The maximum possible number of non-conflicting activities is 4, and 4 have been chosen.

PARTICIPATION ACTIVITY

18.3.5: ActivitySelection algorithm.

©zyBooks 07/21/23 23:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



- 1) The fireworks show and the night movie both finish at 9 PM, so the sorting algorithm could have swapped the order of the 2. If the 2 were swapped, the number of chosen activities would not be affected.



- True
- False

2) Changing snorkeling's _____ would cause snorkeling to be added to the chosen activities.

- start time from 3 PM to 4 PM
- finish time from 5 PM to 4 PM

3) Regardless of any changes to the activity list, the activity with the _____ will always be in the result.

- earliest start time
- earliest finish time
- longest length

©zyBooks 07/21/23 23:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

18.4 Dynamic programming



This section has been set as optional by your instructor.

Dynamic programming overview

Dynamic programming is a problem solving technique that splits a problem into smaller subproblems, computes and stores solutions to subproblems in memory, and then uses the stored solutions to solve the larger problem. Ex: Fibonacci numbers can be computed with an iterative approach that stores the 2 previous terms, instead of making recursive calls that recompute the same term many times over.

PARTICIPATION ACTIVITY

18.4.1: FibonacciNumber algorithm: Recursion vs. dynamic programming.

©zyBooks 07/21/23 23:58 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

Animation content:

undefined

Animation captions:

1. The recursive call hierarchy of FibonacciNumber(4) shows each call made to FibonacciNumber.
2. Several terms are computed more than once.
3. The iterative implementation uses dynamic programming and stores the previous 2 terms at a time.
4. For each iteration, the next term is computed by adding the previous 2 terms. The previous and current terms are also updated for the next iteration.
5. 3 loop iterations are needed to compute FibonacciNumber(4). Because the previous 2 terms are stored, no term is computed more than once.

©zyBooks 07/21/23 23:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

18.4.2: FibonacciNumber implementation.



- 1) If the recursive version of FibonacciNumber(3) is called, how many times will FibonacciNumber(2) be called?

- 1
- 2
- 3



- 2) Which version of FibonacciNumber is faster for large term indices?

- Recursive version
- Iterative version
- Neither



- 3) Which version of FibonacciNumber is more accurate for large term indices?

- Recursive version
- Iterative version
- Neither



- 4) The recursive version of FibonacciNumber has a runtime complexity of $O(\quad)$. What is the runtime complexity of the iterative version?

- $O(\quad)$
- $O(\quad)$

©zyBooks 07/21/23 23:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

O $O(\quad)$

PARTICIPATION ACTIVITY

18.4.3: Dynamic programming.



- 1) Dynamic programming avoids recomputing previously computed results by storing and reusing such results.

O True
 O False

©zyBooks 07/21/23 23:58 169246
Taylor Larrechea
COLORADOCSPB2270Summer2023

- 2) Any algorithm that splits a problem into smaller subproblems is using dynamic programming.

O True
 O False



Longest common substring

The **longest common substring** algorithm takes 2 strings as input and determines the longest substring that exists in both strings. The algorithm uses dynamic programming. An $N \times M$ integer matrix keeps track of matching substrings, where N is the length of the first string and M the length of the second. Each row represents a character from the first string, and each column represents a character from the second string.

An entry at i, j in the matrix indicates the length of the longest common substring that ends at character i in the first string and character j in the second. An entry will be 0 only if the 2 characters the entry corresponds to are not the same.

The matrix is built one row at a time, from the top row to the bottom row. Each row's entries are filled from left to right. An entry is set to 0 if the two characters do not match. Otherwise, the entry at i, j is set to 1 plus the entry in $i - 1, j - 1$.

PARTICIPATION ACTIVITY

18.4.4: Longest common substring algorithm.

©zyBooks 07/21/23 23:58 169246
Taylor Larrechea
COLORADOCSPB2270Summer2023

Animation content:

undefined

Animation captions:

1. Comparing "Look" and "zyBooks" requires a 4×7 matrix.
2. In the first row, 0 is entered for each pair of mismatching characters.
3. In the next row, 'o' matches in 2 entries. In both cases the upper-left value is 0, and 1 is entered into the matrix.
4. Two matches for 'o' exist in the next row as well, with the second having a 1 in the upper-left entry.
5. The character 'k' matches once in the last row and an entry of $2 + 1 = 3$ is entered.
6. The maximum entry in the matrix is the longest common substring's length. The maximum entry's row index is the substring ending index in the first string.

PARTICIPATION ACTIVITY**18.4.5: Longest common substring matrix.**

Consider the matrix below for the two strings "Programming" and "Problem".

		P	r	o	g	r	a	m	m	i	n	g
P	1	0	0	0	0	0	0	0	0	0	0	0
r	0	2	0	0	?	0	0	0	0	0	0	0
o	0	0	?	0	0	0	0	0	0	0	0	0
b	0	0	0	0	0	0	0	0	0	0	0	0
I	0	0	0	0	0	0	0	0	0	0	0	0
e	0	0	0	0	0	0	0	0	0	0	0	0
m	0	0	0	0	0	0	1	?	0	0	0	0

- 1) What should be the value in the green cell?



- 0
- 1
- 2

- 2) What should be the value in the yellow cell?



- 1
- 2
- 3

- 3) What should be the value in the blue cell?



- 0
- 1

2

- 4) What is the longest common substring?

 Pr
 Pro
 mm

©zyBooks 07/21/23 23:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

Longest common substring algorithm complexity

The longest common substring algorithm operates on two strings of length N and M. For each of the N characters in the first string, M matrix entries are computed, making the runtime complexity $O(N \times M)$. Since an $N \times M$ integer matrix is built, the space complexity is also $O(N \times M)$.

Common substrings in DNA

A real-world application of the longest common substring algorithm is to find common substrings in DNA strings. Ex: Common substrings between 2 different DNA sequences may represent shared traits. DNA strings consist of characters C, T, A, and G.

PARTICIPATION ACTIVITY

18.4.6: Finding longest common substrings in DNA.



Animation content:

undefined

Animation captions:

1. Finding common substrings in DNA strings can be used for detecting things such as genetic disorders or for tracing evolutionary lineages.
2. DNA strings are very long, often billions of characters. Dynamic programming is crucial to obtaining a reasonable runtime.
3. Optimizations can lower memory usage by keeping only the following in memory: previous row data and largest matrix entry information.

©zyBooks 07/21/23 23:58 1692462

COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY

18.4.7: Common substrings in DNA.



- 1) Which cannot be a character in a DNA string?

- A
- B
- C

©zyBooks 07/21/23 23:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



- 2) If an animal's DNA string is available, a genetic disorder might be found by finding the longest common substring from the DNA of another animal ____ the disorder.

- with
- without

- 3) When computing row X in the matrix, what information is needed, besides the 2 strings?

- Row X - 1
- Row X + 1
- All rows before X

**PARTICIPATION ACTIVITY**

18.4.8: Longest common substrings - critical thinking.



- 1) If the largest entry in the matrix were the only known value, what could be determined?

- The starting index of the longest common substring within either string
- The character contents of the common substring
- The length of the longest common substring

©zyBooks 07/21/23 23:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



- 2) Suppose only the row and column indices for the largest entry in the matrix were known, and not the value

©zyBooks 07/21/23 23:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

of the largest or any other matrix entry.
What can be determined in O(1)?

- Only the longest common substring's ending index within either string
- The longest common substring's starting and ending indices within either string

Optimized longest common substring algorithm complexity

The longest common substring algorithm can be implemented such that only the previously computed row and the largest matrix entry's location and value are stored in memory. With this optimization, the space complexity is reduced to O(Θ). The runtime complexity remains O(Θ).

©zyBooks 07/21/23 23:58 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

19.1 Quickselect



This section has been set as optional by your instructor.

©zyBooks 07/21/23 23:59 1692462

Taylor Larrechea

Quickselect is an algorithm that selects the smallest element in a list. Ex: Running quickselect on the list (15, 73, 5, 88, 9) with $k = 0$, returns the smallest element in the list, or 5.

For a list with N elements, quickselect uses quicksort's partition function to partition the list into a low partition containing the X smallest elements and a high partition containing the $N-X$ largest elements. The smallest element is in the low partition if $k \leq$ the last index in the low partition, and in the high partition otherwise. Quickselect is recursively called on the partition that contains the element. When a partition of size 1 is encountered, quickselect has found the smallest element.

Quickselect partially sorts the list when selecting the smallest element.

The best case and average runtime complexity of quickselect are both $O(N)$. In the worst case, quickselect may sort the entire list, resulting in a runtime of $O(N^2)$.

Figure 19.1.1: Quickselect algorithm.

```
// Selects kth smallest element, where k is 0-based
Quickselect(numbers, first, last, k) {
    if (first >= last)
        return numbers[first]

    lowLastIndex = Partition(numbers, first, last)

    if (k <= lowLastIndex)
        return Quickselect(numbers, first, lowLastIndex,
k)
    return Quickselect(numbers, lowLastIndex + 1, last,
k)
}
```

©zyBooks 07/21/23 23:59 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

PARTICIPATION
ACTIVITY

19.1.1: Quickselect.



- 1) Calling quickselect with argument k equal to 1 returns the smallest element in the list.

True



False

- 2) The following function produces the same result as quickselect, albeit with a different runtime complexity.

```
Quickselect(numbers, first,
           last, k) {
    Quicksort(numbers, first,
               last)
    return numbers[k]
}
```

©zyBooks 07/21/23 23:59 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

 True False

- 3) Given $k = 4$, if the quickselect call

`Partition(numbers, 0, 10)`
returns 4, then the element being selected is in the low partition.

 True False**CHALLENGE ACTIVITY**

19.1.1: Quickselect.

489394.3384924.qx3zqy7

Start

What is returned when running quickselect on (62, 13, 74, 20, 55, 80, 57) with $k = 5$?

Ex: 10

©zyBooks 07/21/23 23:59 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

[Check](#)[Next](#)

19.2 Bucket sort

©zyBooks 07/21/23 23:59 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023



This section has been set as optional by your instructor.

Bucket sort is a numerical sorting algorithm that distributes numbers into buckets, sorts each bucket with an additional sorting algorithm, and then concatenates buckets together to build the sorted result. A **bucket** is a container for numerical values in a specific range. Ex: All numbers in the range 0 to 49 may be stored in a bucket representing this range. Bucket sort is designed for arrays with non-negative numbers.

Bucket sort first creates a list of buckets, each representing a range of numerical values. Collectively, the buckets represent the range from 0 to the maximum value in the array. For _____ buckets and a maximum value of _____, each bucket represents _____ values. Ex: For 10 buckets and a maximum value of 49, each bucket represents a range of _____ = 5 values; the first bucket will hold values ranging from 0 to 4, the second bucket 5 to 9, and so on. Each array element is placed in the appropriate bucket.

The bucket index is calculated as _____ . Then, each bucket is sorted with an additional sorting algorithm. Lastly, all buckets are concatenated together in order, and copied to the original array.

Figure 19.2.1: Bucket sort algorithm.

©zyBooks 07/21/23 23:59 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

```
BucketSort(numbers, numbersSize, bucketCount) {
    if (numbersSize < 1)
        return

    buckets = Create list of bucketCount buckets

    // Find the maximum value
    maxValue = numbers[0]
    for (i = 1; i < numbersSize; i++) {
        if (numbers[i] > maxValue)
            maxValue = numbers[i]
    }

    // Put each number in a bucket
    for each (number in numbers) {
        index = floor(number * bucketCount / (maxValue +
1))
        Append number to buckets[index]
    }

    // Sort each bucket
    for each (bucket in buckets)
        Sort(bucket)

    // Combine all buckets back into numbers list
    result = Concatenate all buckets together
    Copy result to numbers
}
```

©zyBooks 07/21/23 23:59 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

PARTICIPATION ACTIVITY**19.2.1: Bucket sort.**

Suppose BucketSort is called to sort the list (71, 22, 99, 7, 14), using 5 buckets.

- 1) 71 and 99 will be placed into the same bucket.

- True
- False



- 2) No bucket will have more than 1 number.

- True
- False



- 3) If 10 buckets were used instead of 5, no bucket would have more than 1 number.

©zyBooks 07/21/23 23:59 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023



- True
 - False
-

Bucket sort terminology

©zyBooks 07/21/23 23:59 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023

The term "bucket sort" is sometimes used to refer to a category of sorting algorithms, instead of a specific sorting algorithm. When used as a categorical term, bucket sort refers to a sorting algorithm that places numbers into buckets based on some common attribute, and then combines bucket contents to produce a sorted array.

©zyBooks 07/21/23 23:59 1692462

Taylor Larrechea

COLORADOCSPB2270Summer2023