COP 4530 - Team Assignments

The purpose of this assignment is two-fold. I want to give you experience working with data structures that we did not have time to cover in class. I also want to give you an experience working as a part of a software development team.

Let us both agree from the start: teams are horrible. You could get this work done much easier / quicker / better if I allowed you to just work on it by yourself. However, sadly, that is not the way that the real world works. Instead, we work on teams and it is up to us to draw the best qualities out of the people that we have been thrown together with.

Your team are going to be special. There will be three roles that will have to be assumed by different members of your team. Dr. Anderson does not care who plays what role, just that all of you agree to take on one of the three different roles.

The roles are as follows:

> **Requirements Generator**: This person will not be doing any programming; however, they do have to get their work done before anyone else can start. The requirements generator will read over the task that the team has been assigned. This person is then responsible for creating requirements that will be implemented by the developers. What makes this task challenging is that we are going to assume that the requirements generator does not know how to write Python code. This means that the requirements are going to have to be at a higher level than code. What will the inputs look like? What will the output look like? How does the data have to be processed? These are the types of questions that the team's requirements have to answer.
>
> **2 Developers**: The developers are the people who will actually write the code for the team. These people will take the requirements and attempt to implement them. If something is unclear or wrong, they will go back to the requirements generator and ask them to make a change. If they have a great idea that impacts the requirements, they will go back to the requirements generator and ask them to make a change. In the end, the code that is produced should do no more or no less than was requested by the team's requirements. This development team will have to agree on how to divide up the programming task.
>
> **Tester**: The tester is going to be responsible for testing the team's code and making sure that it works in all situations. This means that they can get started once the requirements generator is done with their work. Based on the requirements they can determine what inputs to use, what outputs are expected. When they have the code they should try to overload it, starve it, and give it bad data to see what happens. When the code does not behave correctly, they can go back to the developer and request that changes get made. They can ask the developer to provide them with a special testing API if so needed.

<u>Note</u>: If your team decides that they only need one developer, you can then have either two requirements generators or two testers. However, what each of these people did as part of the project is going to have to be clearly spelled out.

Each team will turn in both the code, input data, and a slide presentation. We will be double checking to make sure that your code works. The slide presentation will be presented in class. The team will have 15 minutes to present their results. Each member of the team will discuss what they did as a part of the project and how they interacted with the other members of their team. How the program works can also be discussed if time allows. Your grade will be based on your team's in-class presentation.

Teams Alaska & Team South Dakota

Cyclic doubly linked list

Requirements

In this project, you will implement two classes:

1. Cyclic doubly linked lists: `Cyclic_double_list`, and
2. Doubly linked nodes: `Double_node`.

A cyclic doubly linked list with three nodes is shown in Figure 1. The empty cyclic doubly linked list is shown in Figure 2.

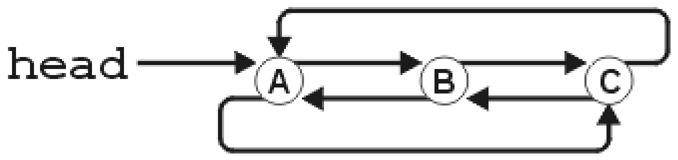

Figure 1. A cyclic doubly linked list and three nodes.

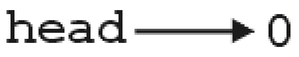

Figure 2. An empty cyclic doubly linked list.

Class Specification

UML Class Diagram

Cyclic_double_list
- list_head:Double_node - list_size:Integer
+ create():Cyclic_double_list + create(in cdl:Cyclic_double_list):Cyclic_double_list + size():Integer + empty():Boolean + front():Type

```
+ back():Type
+ head():Double_node
+ count( in obj:Type ):Integer
+ swap( inout list:Cyclic_double_list )
+ =( in rhs:Cyclic_double_list ):Cyclic_double_list
+ push_front( in obj:Type )
+ push_back( in obj:Type )
+ pop_front():Type
+ pop_back():Type
+ erase( in obj:Type ):Integer
+ destroy()
```

Description

This class stores a finite list of *n* (zero or more) elements stored in doubly linked nodes. If there are zero elements in the list, the list is said to be *empty*. Each element is stored in an instance of the `Double_node<Type>` class. If the list is empty, the head pointer points to `nullptr`. Otherwise, the head pointer points to the first node, the next pointer of the *i*th node ($1 \le i < n$) points to the ($i + 1$)st node, the next pointer of the *n*th node points to the first node, the previous pointer of the *i*th node ($2 \le i \le n$) points to the ($i - 1$)st node, the previous pointer of the first node points to the *n*th node.

Member Variables

The two member variables are:

- A pointer to a `Double_node<Type>` referred to as the *head pointer*, and
- An integer referred to as the *list size* which equals the number of elements in the list.

Member Functions

Constructors

`Cyclic_double_list()`

This constructor sets all member variables to `0` or `nullptr`, as appropriate. (**O**(1))

Destructor

The destructor must delete each of the nodes in the linked list. (**O**(*n*))

Copy Constructor

The copy constructor must create a new cyclic doubly linked list with a copy of all of the nodes within the linked list with the elements stored in the same order. Once a copy is made, any change to the original linked list must not affect the copy. (**O**(*n*))

Accessors

This class has six accessors:

`int size() const;`

Returns the number of items in the list. (**O**(1))

`bool empty() const;`

Returns `true` if the list is empty, `false` otherwise. (**O**(1))

`Type front() const;`

Retrieves the object stored in the node pointed to by the head pointer. This function throws a `underflow` if the list is empty. (**O**(1))

`Type back() const;`

Retrieves the object stored in the last node in the list (the node previous to the head). This function throws a `underflow` if the list is empty. (**O**(1))

`Double_node<Type> *head() const;`

Returns the head pointer. (**O**(1))

`int count( Type const & ) const;`

Returns the number of nodes in the linked list storing a value equal to the argument. (**O**(*n*))

Mutators

This class has seven mutators [In computer science, a mutator method is a method used to control changes to a variable]:

`void swap( Cyclic_double_list & );`

The swap function swaps all the member variables of this linked list with those of the argument. (**O**(1))

`Cyclic_double_list &operator=( Cyclic_double_list & );`

The assignment operator makes a copy of the argument and then swaps the member variables of this cyclic doubly linked list with those of the copy. (**O**($n_{\text{lhs}} + n_{\text{rhs}}$))

`void push_front( Type const & );`

Creates a new `Double_node<Type>` storing the argument, the next pointer of which is set to the current head pointer, and the previous pointer of which is set to the node previous to the current head pointer. The previous pointer of what was the first node as well as the next pointer of the last node in the list are also set to the new node. The head pointer is set to this new node. If the list was originally empty, both the next and previous pointers of the new node are set to itself. (**O**(1))

`void push_back( Type const & );`

Similar to `push_front`, this places a new node at the back of the list (it does the same as `push_front` but does not update the head pointer). (**O**(1))

`Type pop_front();`

Delete the node at the front of the linked list and, as necessary, update the head pointer and the previous and next pointers of any other node within the list. Return the object stored in the node being popped. Throw an `underflow` exception if the list is empty. (**O**(1))

```
Type pop_back();
```

Similar to `pop_front`, delete the last node in the list. This function throws a `underflow` if the list is empty. (**O**(1))

```
int erase( Type const & );
```

Delete the first node (from the front) in the linked list that contains the object equal to the argument (use == to to test for equality with the retrieved element). As necessary, update the head pointer and the previous and next pointers of any other node within the list. Return the number of nodes that were deleted. (**O**(*n*))

Teams Oregon & Texas

Doubly linked list

Requirements

In this project, you will implement two classes:

1. Doubly linked lists: `Double_list`, and
2. Doubly linked nodes: `Double_node`.

A doubly linked list with three nodes is shown in Figure 1. The empty doubly linked list is shown in Figure 2.

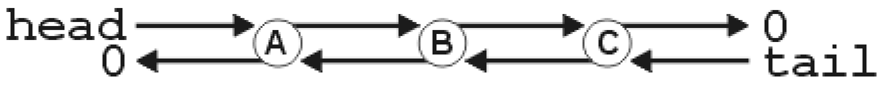

Figure 1. A doubly linked list with a header and three nodes.

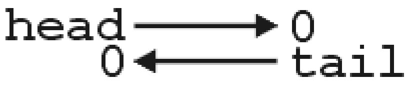

Figure 2. An empty doubly linked list.

Class Specifications

UML Class Diagram

Double_list
- list_head:Double_node - list_tail:Double_node - list_size:Integer
+ create():Double_list + create(in dl:Double_list):Double_list + size():Integer + empty():Boolean + front():Type + back():Type + head():Double_node

+ tail():Double_node + count(in obj:Type):Integer + swap(inout list:Double_list) + =(in rhs:Double_list):Double_list + push_front(in obj:Type) + push_back(in obj:Type) + pop_front():Type + pop_back():Type + erase(in obj:Type):Integer + destroy()

Description

This class stores a finite list of *n* (zero or more) elements stored in doubly linked nodes. If there are zero elements in the list, the list is said to be *empty*. Each element is stored in an instance of the `Double_node<Type>` class. If the list is empty, the head and tail pointers are assigned `nullptr`. Otherwise, the head pointer points to the first node, the tail pointer points to the *n*th node, the next pointer of the *i*th node ($1 \leq i < n$) points to the ($i + 1$)st node, the next pointer of the *n*th is assigned `nullptr`, the previous pointer of the *i*th node ($2 \leq i \leq n$) points to the ($i - 1$)st node, and the previous pointer of the first node is assigned `nullptr`.

Member Variables

The three member variables are:

- Two pointers to `Double_node<Type>` objects, referred to as the *head pointer* and *tail pointer*, respectively, and
- An integer referred to as the *list size* which equals the number of elements in the list.

Member Functions

Constructors

`Double_list()`

This constructor sets all member variables to `0` or `nullptr`, as appropriate. (**O**(1))

Destructor

The destructor must delete each of the nodes in the linked list. (**O**(*n*))

Copy Constructor

The copy constructor must create a new doubly linked list with a copy of all of the nodes within the linked list with the elements stored in the same order. Once a copy is made, any change to the original linked list must not affect the copy. (**O**(*n*))

Accessors

This class has seven accessors:

`int size() const;`
    Returns the number of items in the list. (**O**(1))

`bool empty() const;`
    Returns `true` if the list is empty, `false` otherwise. (**O**(1))

`Type front() const;`
    Retrieves the object stored in the node pointed to by the head pointer. This function throws a `underflow` if the list is empty. (**O**(1))

`Type back() const;`
    Retrieves the object stored in the node pointed to by the tail pointer. This function throws a `underflow` if the list is empty. (**O**(1))

`Double_node<Type> *head() const;`
    Returns the head pointer. (**O**(1))

`Double_node<Type> *tail() const;`
    Returns the tail pointer. (**O**(1))

`int count( Type const & ) const;`
    Returns the number of nodes in the linked list storing a value equal to the argument. (**O**(*n*))

Mutators

This class has seven mutators [In computer science, a **mutator** method is a method used to control changes to a variable]:

`void swap( Double_list & );`
    The swap function swaps all the member variables of this linked list with those of the argument. (**O**(1))

`Double_list &operator=( Double_list & );`
    The assignment operator makes a copy of the argument and then swaps the member variables of this doubly linked list with those of the copy. (**O**($n_{\text{lhs}} + n_{\text{rhs}}$))

`void push_front( Type const & );`
    Creates a new `Double_node<Type>` storing the argument, the next pointer of which is set to the current head pointer and the previous pointer is set to `nullptr`. The head pointer and the previous pointer of what was the first node is set to the new node. If the list was originally empty, the tail pointer is set to point to the new node. (**O**(1))

`void push_back( Type const & );`
    Similar to `push_front`, this places a new node at the back of the list. (**O**(1))

`Type pop_front();`
    Delete the node at the front of the linked list and, as necessary, update the head and tail pointers and the previous pointer of any other node within the list. Return the object

stored in the node being popped. Throw an `underflow` exception if the list is empty. (**O**(1))

`Type pop_back();`

Similar to `pop_front`, delete the last node in the list. This function throws a `underflow` if the list is empty. (**O**(1))

`int erase( Type const & );`

Delete the first node (from the front) in the linked list that contains the object equal to the argument (use == to to test for equality with the retrieved element). As necessary, update the head and tail pointers and the previous and next pointers of any other node within the list. Return the number of nodes that were deleted (0 or 1). (**O**(*n*))

Team California

Sorted doubly linked list

Requirements

In this project, you will implement two classes:

1. Sorted doubly linked lists: `Sorted_double_list`, and
2. Doubly linked nodes: `Double_node`.

A doubly linked list with three nodes is shown in Figure 1. The empty doubly linked list is shown in Figure 2.

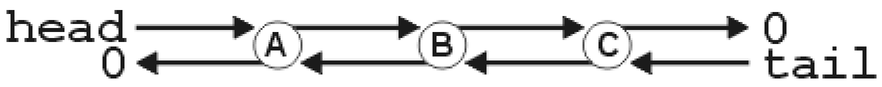

Figure 1. A doubly linked list with a header and three nodes.

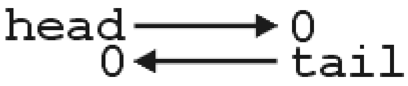

Figure 2. An empty doubly linked list.

Class Specifications

UML Class Diagram

Sorted_double_list
- list_head:Double_node - list_tail:Double_node - list_size:Integer
+ create():Sorted_double_list + create(in dl:Sorted_double_list):Sorted_double_list + size():Integer + empty():Boolean + front():Type + back():Type + head():Double_node

|
|---|
| + tail():Double_node
+ count(in obj:Type):Integer
+ swap(inout list:Sorted_double_list)
+ =(in rhs:Sorted_double_list):Sorted_double_list
+ insert(in obj:Type)
+ pop_front():Type
+ pop_back():Type
+ erase(in obj:Type):Integer
+ destroy() |

Description

This class stores a finite list of n (zero or more) linearly ordered elements in that order stored in doubly linked nodes. If there are zero elements in the list, the list is said to be *empty*. Each element is stored in an instance of the `Double_node<Type>` class. If the list is empty, the head and tail pointers are assigned `nullptr`. Otherwise, the head pointer points to the first node, the tail pointer points to the nth node, the next pointer of the ith node ($1 \leq i < n$) points to the (i + 1)st node, the next pointer of the nth is assigned `nullptr`, the previous pointer of the ith node ($2 \leq i \leq n$) points to the (i − 1)st node, and the previous pointer of the first node is assigned `0`.

Member Variables

The three member variables are:

- Two pointers to `Double_node<Type>` objects, referred to as the *head pointer* and *tail pointer*, respectively, and
- An integer referred to as the *list size* which equals the number of elements in the list.

Member Functions

Constructors

`Sorted_double_list()`

This constructor sets all member variables to `0` or `nullptr`, as appropriate. (**O**(1))

Destructor

The destructor must delete each of the nodes in the linked list. (**O**(n))

Copy Constructor

The copy constructor must create a new doubly linked list with a copy of all of the nodes within the linked list with the elements stored in the same order. Once a copy is made, any change to the original linked list must not affect the copy. (**O**(*n*) and note that using `insert` will result in an **O**(n^2) run-time)

Accessors

This class has seven accessors:

`int size() const;`
    Returns the number of items in the list. (**O**(1))
`bool empty() const;`
    Returns `true` if the list is empty, `false` otherwise. (**O**(1))
`Type front() const;`
    Retrieves the object stored in the node pointed to by the head pointer. This function throws a `underflow` if the list is empty. (**O**(1))
`Type back() const;`
    Retrieves the object stored in the node pointed to by the tail pointer. This function throws a `underflow` if the list is empty. (**O**(1))
`Double_node<Type> *head() const;`
    Returns the head pointer. (**O**(1))
`Double_node<Type> *tail() const;`
    Returns the tail pointer. (**O**(1))
`int count( Type const & ) const;`
    Returns the number of nodes in the linked list storing a value equal to the argument. (**O**(*n*))

Mutators

This class has seven mutators [In computer science, a **mutator** method is a method used to control changes to a variable]:

`void swap( Sorted_double_list & );`
    The swap function swaps all the member variables of this linked list with those of the argument. (**O**(1))
`Sorted_double_list &operator=( Sorted_double_list & );`
    The assignment operator makes a copy of the argument and then swaps the member variables of this doubly linked list with those of the copy. (**O**($n_{\text{lhs}} + n_{\text{rhs}}$))
`void insert( Type const & );`
    Creates a new `Double_node<Type>` storing the argument, placing it into the correct location in the linked list such that the element in the previous node (if any) is less than or equal to the element stored in the current node, and that element is less than or equal to the element stroed in the next node. The head and tail pointers may have be updated if the new node is placed at the head or tail of the linked list. (**O**(*n*))
`Type pop_front();`
    Delete the node at the front of the linked list and, as necessary, update the head and tail pointers and the previous pointer of any other node within the list. Return the object

stored in the node being popped. Throw an `underflow` exception if the list is empty. (**O**(1))

`Type pop_back();`

Similar to `pop_front`, delete the last node in the list. This function throws a `underflow` if the list is empty. (**O**(1))

`int erase( Type const & );`

Delete the first node (from the front) in the linked list that contains the object equal to the argument (use == to to test for equality with the retrieved element). As necessary, update the head and tail pointers and the previous and next pointers of any other node within the list. Return the number of nodes that were deleted. (**O**(*n*))

Team Montana

Sorted singly linked sentinel list

Requirements

In this project, you will implement two classes:

1. Sorted singly linked lists with a sentinel: `Sorted_sentinel_list`, and
2. Singly linked nodes: `Single_node`.

A singly linked list with a sentinel and three nodes is shown in Figure 1. The empty singly linked list with a sentinel is shown in Figure 2, the node marked **S** being the sentinel node.

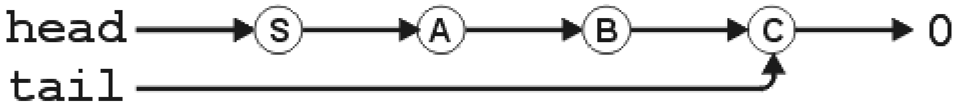

Figure 1. A singly linked list with a sentinel and three nodes.

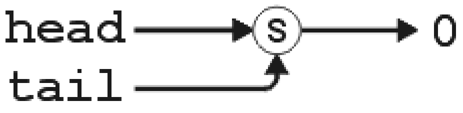

Figure 2. An empty singly linked list with a sentinel.

Class Specifications

UML Class Diagram

Sorted_sentinel_list
- list_head:Single_node
- list_tail:Single_node
- list_size:Integer
+ create():Sorted_sentinel_list
+ create(in sl:Sorted_sentinel_list):Sorted_sentinel_list
+ size():Integer
+ empty():Boolean
+ front():Type

+ back():Type
+ head():Single_node
+ tail():Single_node
+ count(in obj:Type):Integer
+ swap(inout list:Sorted_sentinel_list)
+ =(in rhs:Sorted_sentinel_list):Sorted_sentinel_list
+ insert(in obj:Type)
+ push_back(in obj:Type)
+ pop_front():Type
+ erase(in obj:Type):Integer
+ destroy()

Description

This class stores a finite list of *n* (zero or more) linearly ordered elements in that order stored in singly linked nodes. If there are zero elements in the list, the list is said to be *empty*. Each element is stored in an instance of the `Single_node<Type>` class. At all times, the head pointer points to a *sentinel* node. If the list is empty, the tail pointer points to the sentinel node.

Member Variables

The three member variables are:

- Two pointers to `Single_node<Type>` objects, referred to as the *head pointer* and *tail pointer*, respectively, and
- An integer referred to as the *list size* which equals the number of elements in the list.

Member Functions

Constructors

`Sorted_sentinel_list()`

The constructor creates an instance of a `Single_node<Type>` (called the *sentinel*). The value stored in this node is not important, you can use the default value or whatever value you want. The next pointer of the sentinel should be `nullptr`. The head and tail pointers are set to point at the sentinel. The list size is set to `0`. (**O**(1))

Destructor

The destructor must delete each of the nodes in the list including the sentinel. (**O**(*n*))

Copy Constructor

The copy constructor must create a new singly linked list with a copy of all of the nodes within the linked list with the elements stored in the same order. Once a copy is made, any change to the original linked list must not affect the copy. (**O**(n) and note that using `insert` will result in an **O**(n^2) run-time)

Accessors

This class has seven accessors:

`int size() const;`
: Returns the number of items in the list. (**O**(1))

`bool empty() const;`
: Returns `true` if the list is empty, `false` otherwise. (**O**(1))

`Type front() const;`
: Retrieves the object stored in the node pointed to by the next pointer of the sentinel. This function throws a `underflow` if the list is empty. (**O**(1))

`Type back() const;`
: Retrieves the object stored in the node pointed to by the tail pointer. This function throws a `underflow` if the list is empty. (**O**(1))

`Single_node<Type> *head() const;`
: Returns the head pointer. (**O**(1))

`Single_node<Type> *tail() const;`
: Returns the tail pointer. (**O**(1))

`int count( Type const & ) const;`
: Returns the number of nodes in the linked list storing a value equal to the argument. (**O**(n))

Mutators

This class has six mutators [In computer science, a **mutator** method is a method used to control changes to a variable]:

`void swap( Sorted_sentinel_list & );`
: The swap function swaps all the member variables of this linked list with those of the argument. (**O**(1))

`Sorted_sentinel_list &operator=( Sorted_sentinel_list & );`
: The assignment operator makes a copy of the argument and then swaps the member variables of this sorted sentinel list with those of the copy. (**O**($n_{\text{lhs}} + n_{\text{rhs}}$))

`void insert( Type const & );`
: Creates a new `Single_node<Type>` storing the argument, placing it into the correct location in the linked list such that the element in the previous node (if any) is less than or equal to the element stored in the current node, and that element is less than or equal to the element stroed in the next node. (**O**(n))

`Type pop_front();`

Delete the node at the front of the linked list and, as necessary, update the tail pointer and the next pointer of the sentinel. Return the object stored in the node being popped. Throw an `underflow` exception if the list is empty. (**O**(1))

`int erase( Type const & );`

Delete all nodes (other than the sentinals) in the linked list that contains the object equal to the argument (use == to to test for equality with the retrieved element). As necessary, update the tail pointer and the next pointer of any other node (including possibly the sentinel) within the list. Return the number of nodes that were deleted. (**O**(*n*))

Team New Mexico

Dynamic stack

Requirements:

In this project, you will implement one class:

1. Dynamice Stack: `Dyanamic_stack`.

A stack stores elements in an ordered list and allows insertions and deletions at one end of the list in **O**(1) time.

The elements in this stack are stored in an array. The size of the array may be changed depending on the number of elements currently stored in the array, according to the following two rules:

- If an element is being inserted into a stack where the array is already full, the size of the array is doubled.
- If, after removing an element from a stack where the number of elements is 1/4 the size of the array, then the size of the array is halved. The size of the array may not be reduced below the initially specified size.

Runtime:

The amortized run time of each member function is specified in parentheses at the end of the description.

Class Specifications:

`Dyanamic_stack`

Description

A class which implements a stack using an array. The size of the array may be changed dynamically after insertions or deletions. For run-time requirements, the number of elements in the stack is *n*.

Member Variables

The class at least four members:

- A pointer to an instance of Type, `Type *array`, to be used as an array,
- A counter `int count`,
- The initial size of the array, `int initial_size`, and
- The current size of the array, `int array_size`.

Member Functions

Constructors

```
Dyanamic_stack( int n = 10 )
```

The constructor takes as an argument the initial size of the array and allocates memory for that array. The default number of entries is 10. Other class members are assigned as appropriate.

Destructor

```
~Dyanamic_stack()
```

The destructor deletes the memory allocated for the array.

Accessors

This class has four accessors:

`Type top() const`
: Return the object at the top of the stack. It may throw a `underflow` exception). (**O**(1))

`int size() const`
: Returns the number of elements currently stored in the stack. (**O**(1))

`bool empty() const`
: Returns `true` if the stack is empty, `false` otherwise. (**O**(1))

`int capacity() const`
: Returns the current size of the array. (**O**(1))

Mutators

This class has three mutators [In computer science, a **mutator** method is a method used to control changes to a variable]:

`void push( Type const & )`
: Insert the new element at the top of the stack. If the array is filled, the size of the array is doubled. (**O**(1) on average)

`Type pop()`

Removes the element at the top of the stack. If, after the element is removed, the array is 1/4 full and the array size is greater than the initial size, the size of the array is halved. This may throw a `underflow` exception. (**O**(1) on average)

`void clear()`

Removes all the elements in the stack. The array is resized to the initial size. (**O**(1))

Team Arizona

Dynamic queue

Requirements:

In this project, you will implement one class:

1. Dynamic Queue: `Dynamic_queue`.

A queue stores objects in an ordered list and allows insertions at one end and deletions from the other end of the list in **O**(1) time.

The objects in this queue are stored in an array. The capacity of the array may be changed depending on the number of objects currently stored in the array, according to the following two rules:

- If an object is being inserted into a queue where the array is already full, the capacity of the array is doubled.
- If, after removing an object from a queue where the number of objects is one-quarter (1/4) the capacity of the array, then the capacity of the array is halved. The capacity of the array may not be reduced below the initially specified capacity.

Runtime:

The amortized run time of each member function is specified in parentheses at the end of the description. Projects which do not satisfy the run time requirements will be required to resubmit.

Class Specifications:

`Dynamic_queue`

Description

A class which implements a queue using an array. The capacity of the array may be changed dynamically after insertions or deletions. For run-time requirements, the number of objects in the queue is *n*.

Member Variables

The class at least six suggested member variables:

- A pointer to an instance of Type, `Type *array`, to be used as an array,
- A head index `int ihead`,
- A tail index `int itail`,
- A counter `int entry_count`,
- The initial capacity of the array, `int initial_capacity`, and
- The current capacity of the array, `int array_capacity`.

You may chose to use these or use whatever other member variables you want. You do not have to use all of these member variables—it is possible to implement this class using two of the three integer member variables; however, this requires more computation for the individual member functions.

Member Functions

Constructors

```
Dynamic_queue( int n = 10 )
```

The constructor takes as an argument the initial capacity of the array and allocates memory for that array. If the argument is either 0 or a negative integer, set the initial capacity of the array to 1. The default initial capacity of the array is 10. Other member variables are assigned as appropriate.

Destructor

```
~Dynamic_queue()
```

The destructor deletes the memory allocated for the array.

Copy Constructor

```
Dynamic_queue( Dynamic_queue const & )
```

The copy constructor creates a new instance of the queue. (**O**(n))

Accessors

This class has four accessors:

```
Type head() const
```

Return the object at the head of the queue (the object that would be removed by `Type dequeue()`). It may throw a `underflow` exception). (**O**(1))

`int size() const`

Returns the number of objects currently stored in the queue. (**O**(1))

`bool empty() const`

Returns `true` if the queue is empty, `false` otherwise. (**O**(1))

`int capacity() const`

Returns the current capacity of the queue. (**O**(1))

Mutators

This class has five mutators [In computer science, a **mutator** method is a method used to control changes to a variable]:

`void swap( Dynamic_queue & );`

The swap function swaps all the member variables of this queue with those of the argument. (**O**(1))

`Dynamic_queue &operator=( Dynamic_queue & );`

The argument is a copy of the right-hand side of the operator. Swap the member variables of this with that copy.

`void enqueue( Type const & )`

Insert the argument at the tail of the queue. If the array is full before the argument is placed into the queue, the capacity of the array is first doubled. (**O**(1) on average)

`Type dequeue()`

Removes the object at the head of the queue. If, after the object is removed, the array is one-quarter (1/4) full and the array capacity is greater than the initial capacity, the capacity of the array is halved. This will throw the `underflow` exception if the queue is empty. (**O**(1) on average)

`void clear()`

Empties the queue by resetting the member variables. The array is resized to the initial capacity. (**O**(1))

Team Nevada

Linked stack

Requirements:

In this project, you will implement one class:

1. A linked stack class: `Linked_stack`.

Runtime:

The run time of each member function is specified in parentheses at the end of the description.

Class Specifications:

`Linked_stack`

Description

A class which implements a linked stack (in fact, two stacks) which has the specified behaviours. For run-time requirements, the number of elements in the stack is *n*.

Member Variables

The class has four member variables:

- A linked list of pointers to arrays of `Type`: `Single_list<Type *> array_list` (or whatever linked list you created in Project 1),
- An integer storing the stack size, and
- One integer `itop`.

Each array in the linked list will have a capacity of eight. When the stack is empty, the linked list is empty. As objects are placed into the stack, they are placed into the arrays that will be stored in the linked list. As necessary, additional arrays are added to the linked list. For example, suppose we begin with an empty stack and then push five times and pop once. The result will appear as shown in Figure 1 where the member variables `itop = 3`.

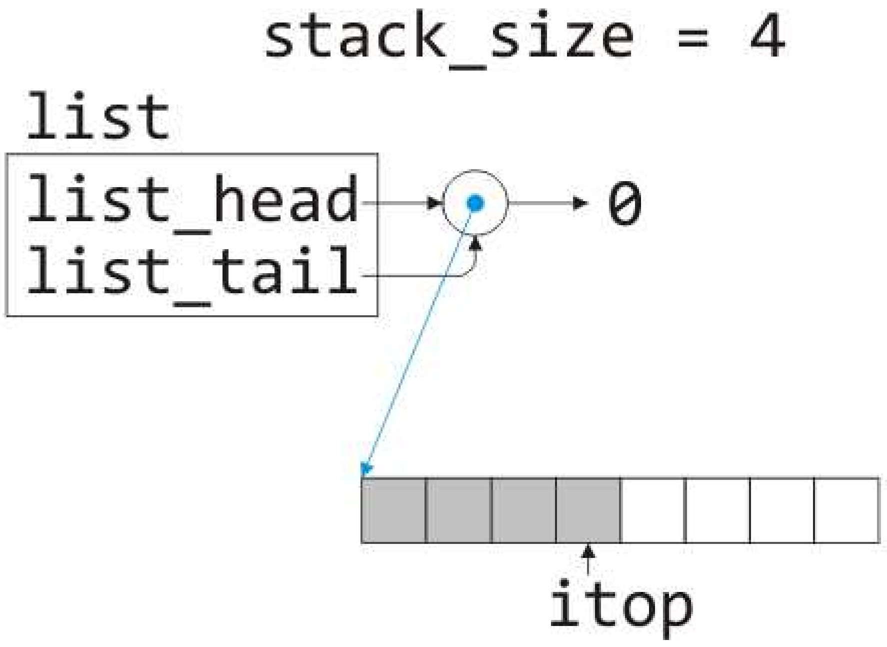

Figure 1. The stack after five pushes and one pop.

Now, suppose that there is a sequence of fifteen pushes: when the array at the back of the linked list is filled, a new array is pushed onto the back of the linked list, and new entries are pushed into the array pointed to by that node. When this array also becomes filled, another array will be pushed onto the linked list. The member variable `itop` keeps track of the location in the top in the front array. As shown in Figure 2, `itop = 2` and the linked list has three entries.

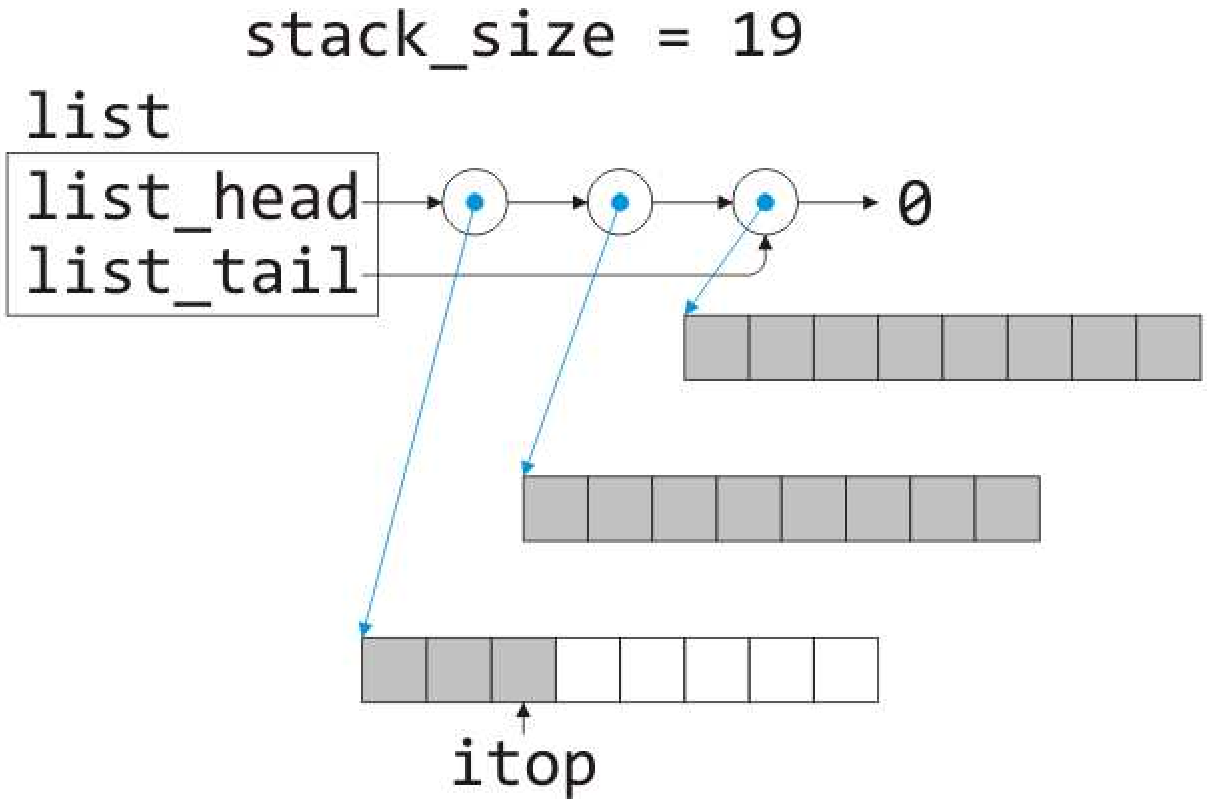

Figure 2. The stack in Figure 1 after another fifteen pushes.

Suppose next there is a sequence of ten pops. After the third pop, the node at the front of the linked list would be popped and the memory for the array it points to would be deallocated and `itop` would be reset to one less than the array capacity. After seve more pops, `itop = 0`, as shown in Figure 3.

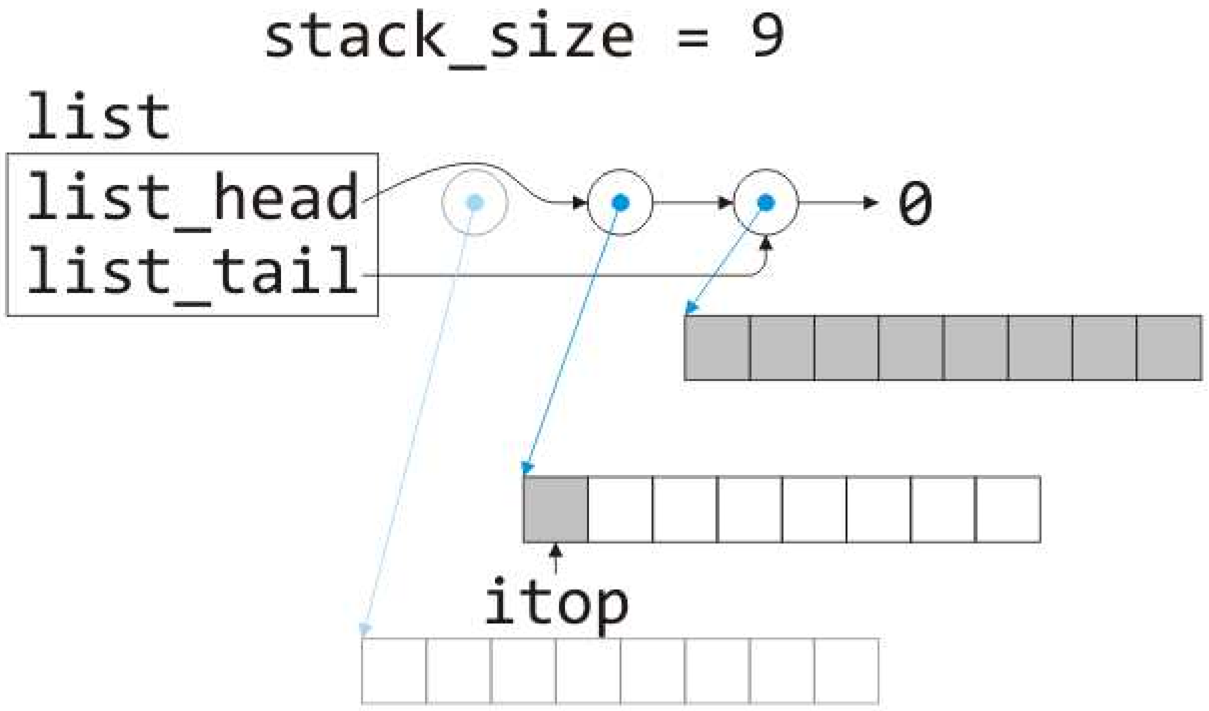

Figure 3. The stack in Figure 2 after ten pops.

Member Functions

Constructors

The default constructor is used.

Copy constructor

Make a complete copy of the linked stack. For each array in the original linked list, a new array must be allocated and the entries copied over.

Destructor

The destructor will have to empty the linked list and deallocate the memory pointed to by the entries of the linked list.

Accessors

This class has three accessors:

`bool empty() const`
    Returns true if the stack is empty. (**O**(1))
`int size() const`
    Returns the number of objects currently in the stack. (**O**(1))

`int list_size() const`

Returns the number of nodes in the linked list data structure. This must be implemented as provided. (**O**(1))

`Type top() const`

Returns the object at the top of the stack. This member function may throw an `underflow` exception. (**O**(1))

Mutators

This class has four mutators [In computer science, a **mutator** method is a method used to control changes to a variable]:

`void swap( Linked_stack & );`

The swap function swaps all the member variables of this stack with those of the argument. (**O**(*n*))

`Linked_stack &operator=( Linked_stack & );`

The assignment operator makes a copy of the argument and then swaps the member variables of this node with those of the copy. (**O**($n_{\text{lhs}} + n_{\text{rhs}}$))

`void push( Type const & )`

Push the argument onto the back of the stack:

- If the stack is empty, allocate memory for a new array with the required capacity, push the address of that array onto the linked list, set both indices to zero and place the new argument at that location. The size of the stack is now one.
- If the back index already points to the last entry of the array, reset it to zero, allocate memory for a new array with the required capacity, push the address of that array onto the linked list, and insert the argument into the first location.
- Otherwise, increment the back index and place the argument at that location.

Increment the stack size. (**O**(1))

`Type pop()`

Pop the top of the stack and decrement the `itop` index. If the top index equals 0, reset it to the array capacity minus one and pop the top of the linked list while deallocating the memory allocated to that array. If the stack is emptied, also pop the front of the linked list while deallocated the memory allocated to that array. This member function may throw a `underflow` exception. (**O**(1))

Team Colorado

Linked queue

Requirements:

In this project, you will implement one class:

1. A linked queue class: `Linked_queue`.

Runtime:

The run time of each member function is specified in parentheses at the end of the description.

Class Specifications:

`Linked_queue`

Description

A class which implements a linked queue (in fact, two stacks) which has the specified behaviors. For run-time requirements, the number of elements in the queue is *n*.

Member Variables

The class has four member variables:

- A linked list of pointers to arrays of `Type`: `Single_list<Type *> array_list` (or whatever linked list you created in Project 1),
- An integer storing the queue size, and
- Two integers `ifront` and `itail`.

Each array in the linked list will have a capacity of eight. When the queue is empty, the linked list is empty. As objects are placed into the queue, they are placed into the arrays that will be stored in the linked list. As necessary, additional arrays are added to the linked list. For example, suppose we begin with an empty queue and then push six times and pop once. The result will appear as shown in Figure 1 where the member variables `ifront = 1` and `iback = 5`.

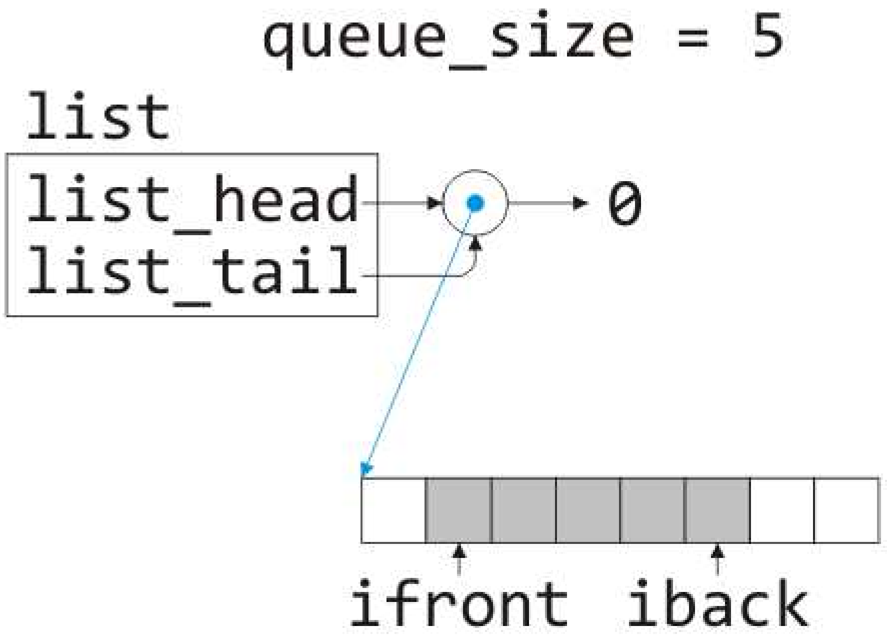

Figure 1. The queue after six pushes and one pop.

Now, suppose that there is a sequence of fourteen pushes: when the array at the back of the linked list is filled, a new array is pushed onto the back of the linked list, and new entries are pushed into the array pointed to by that node. When this array also becomes filled, another array will be pushed onto the linked list. The member variables `ifront` and `iback` keep track of the locations in the first and last entries in their respective arrays. As shown in Figure 2, `ifront = 1` and `itail = 3` and the linked list has three entries.

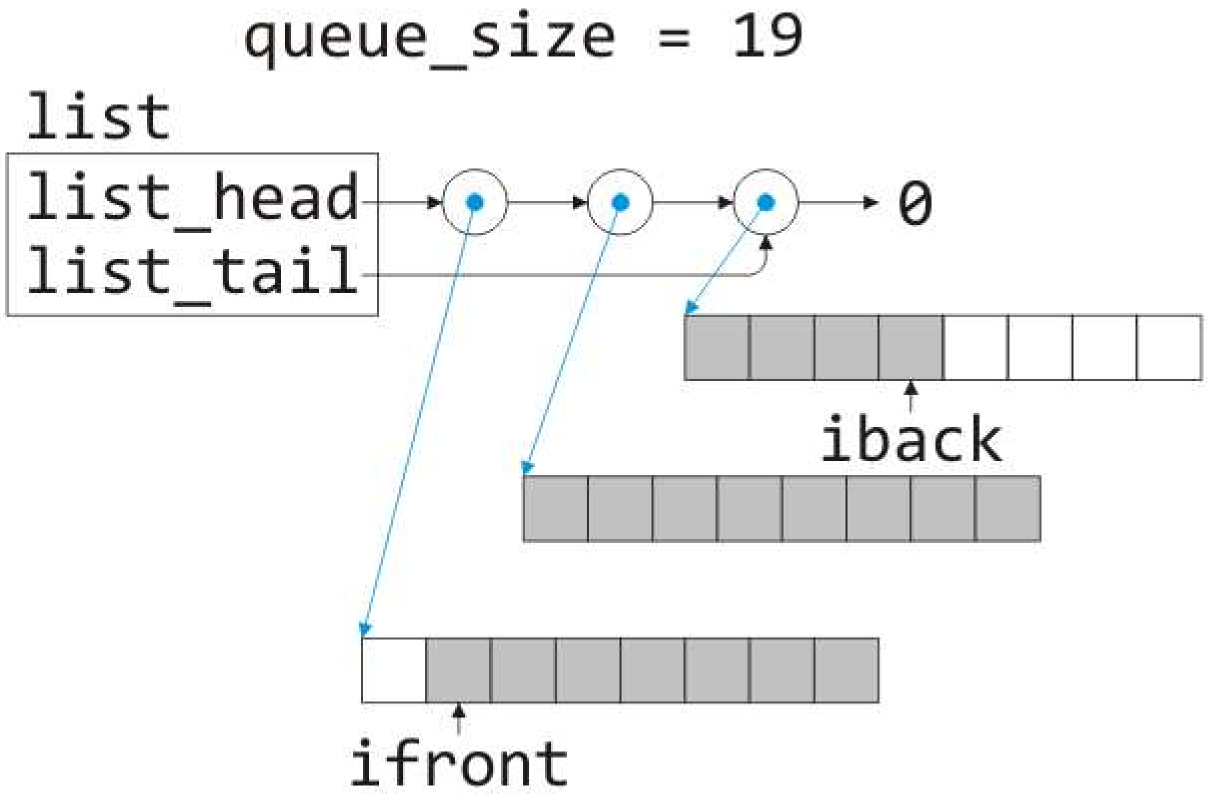

Figure 2. The queue in Figure 1 after another fourteen pushes.

Suppose next there is a sequence of nine pops. After the seventh pop, the node at the front of the linked list would be popped and the memory for the array it points to would be deallocated and `ifront` would be reset to `0`. After two more pops, `ifront = 2`, as shown in Figure 3.

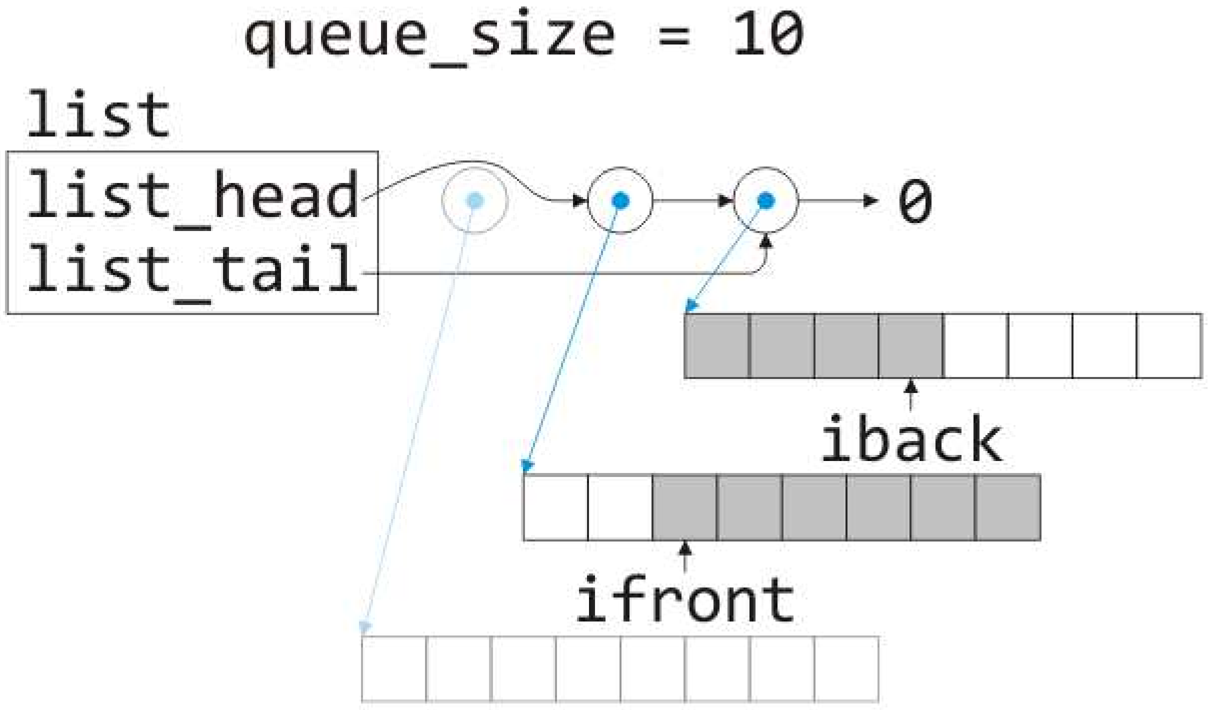

Figure 3. The queue in Figure 2 after nine pops.

Member Functions

Constructors

The default constructor is used.

Copy constructor

Make a complete copy of the linked queue. For each array in the original linked list, a new array must be allocated and the entries copied over.

Destructor

The destructor will have to empty the linked list and deallocate the memory pointed to by the entries of the linked list.

Accessors

This class has three accessors:

`bool empty() const`
  Returns true if the queue is empty. (**O**(1))

`int size() const`
  Returns the number of objects currently in the queue. (**O**(1))

`int list_size() const`

Returns the number of nodes in the linked list data structure. This must be implemented as provided. (**O**(1))

`Type front() const`

Returns the object at the front of the queue. This member function may throw an `underflow` exception. (**O**(1))

Mutators

This class has four mutators [In computer science, a **mutator** method is a method used to control changes to a variable]:

`void swap( Linked_queue & );`

The swap function swaps all the member variables of this stack with those of the argument. (**O**(n))

`Linked_queue &operator=( Linked_queue & );`

The assignment operator makes a copy of the argument and then swaps the member variables of this node with those of the copy. (**O**($n_{\text{lhs}} + n_{\text{rhs}}$))

`void push( Type const & )`

Push the argument onto the back of the queue:

- If the queue is empty, allocate memory for a new array with the required capacity, push the address of that array onto the linked list, set both indices to zero and place the new argument at that location. The size of the queue is now one.
- If the back index already points to the last entry of the array, reset it to zero, allocate memory for a new array with the required capacity, push the address of that array onto the linked list, and insert the argument into the first location.
- Otherwise, increment the back index and place the argument at that location.

Increment the queue size. (**O**(1))

`Type pop()`

Pop the front of the queue and increment the `ifront` index. If the front index equals the aray capacity, reset it to zero and pop the front of the linked list while deallocating the memory allocated to that array. If the queue is emptied, also pop the front of the linked list while deallocated the memory allocated to that array. This member function may throw a `underflow` exception. (**O**(1))

Team Idaho

B-Trees

Requirements

In this project, you will implement two classes:

1. A B+-Tree: `Tree`, and
2. A B+-Tree Node: `B_tree_node`.

Class Specification

UML Class Diagram

B_tree
- tree_root:B_tree_node
+ create():B_tree + size():Integer + empty():Boolean + root():B_tree_node + count(in obj:Type):Integer + insert(in obj:Type) + destroy()

Description

This class stores a finite ordered set of *n* (zero or more) elements stored in B+-tree nodes. If there are zero elements in the tree, the tree is said to be *empty*. Each element is stored in an instance of the `B_tree_node<Type>` class. By default, an empty B+-tree has a single empty B+-tree node, the address of which is stored in `tree_root`. Unlike the singly-linked list class, the B+-tree class, for the most part, simply makes the appropriate function call.

Member Variables

The one class variable is

- One pointer to a `B_tree_node<Type>` object, referred to as the *root*.

Member Functions

Constructors

`B_tree()`

This constructor creates an empty `B_tree_node<Type>` and assigns this to the member variable `tree_root`.

Destructor

The destructor must delete all the nodes in the B+-tree in some manner.

Copy Constructor

There is no copy constructor in this class.

Assignment Operator `=`

There is no assignment operator in this class.

Accessors

This class has four accessors:

`int size() const;`
   Returns the number of items in the tree.

`bool empty() const;`
   Returns `true` if the tree is empty, `false` otherwise. (**O**(1))

`B_tree_node<Type> *root() const;`
   Returns the root pointer. (**O**(1))

`int count( Type const & ) const;`
   Returns `1` if the argument is stored by one of the nodes in the B+-tree and `0` otherwise.

Mutators

This class has one mutator [In computer science, a **mutator** method is a method used to control changes to a variable]:

`void insert( Type const & );`
   Passes the argument to the root B+-tree node. If the `insert` member function of the B+-tree node returns a non-zero pointer, this indicates that the root must split, in which case a new root node is created with the root node as the first sub-tree and the returned pointer as the second sub-tree. You will note that there is a specific B+-tree node constructor to do this.

Team Kansas

Expression Tree

Requirements

In this project, you will implement the class `ExpressionTree`.

An expression tree is a binary tree which stores either:

- An operator and two non-empty sub trees, or
- A value (an integer).

Class Specifications

UML Class Diagram

Expression Tree
- element:Integer - left_tree:ExpressionTree - right_tree:ExpressionTree

+ create(in v:Integer = 0, in l:ExpressionTree = 0, in r:ExpressionTree = 0):ExpressionTree
+ evaluate():Integer
+ in_fix(in parent_op:Integer, in is_left:Boolean)
+ reverse_polish()
+ is_leaf():Boolean
+ destroy()

Description

A class which stores an integer and two pointers to other expression trees. Either a node is full (in which case, it stores a value which identifies the type of operator) or it is a leaf node, in which case it stores an integer. For run time requirements, *n* is the number of nodes in the sub tree defined by this node.

Member Variables

The three member variables are:

- An integer either an integer or an operator ID, and
- Two pointers to `ExpressionTree` objects, referred to as the *left sub-tree* and *right sub-tree*, respectively.

Class Constants

The class has four static constants:

- `ExpressionTree::PLUS`
- `ExpressionTree::MINUS`
- `ExpressionTree::TIMES`
- `ExpressionTree::DIVIDE`

You may use these in creating your expression tree.

Member Functions

Constructors

```
ExpressionTree( int = 0, ExpressionTree * = 0, ExpressionTree * = 0 )
```

This constructor takes three arguments: an integer and two pointers which point to `ExpressionTree` objects. (**O**(1))

Destructor

The destructor deletes the left and right sub trees if they are not `0`.

Accessors

This class has four accessors:

`int evaluate() const`
: If the node is a leaf node, return the value. Otherwise, the node represents an operator, in which case, you perform the operation on the evaluated sub trees. If a division is being performed and the denominator is zero, a `division_by_zero` exception is thrown. (**O**(*n*))

`void in_fix( int, bool ) const`
: If the node is a leaf node, print the node. Otherwise, print the expression using in-fix notation. This is specified on the page for `Expression`. The arguments may be used to pass information from the parent node to a child. (**O**(*n*))

`void reverse_polish() const`
: If the node is a leaf node, print the node. Otherwise, print the left sub-tree, then the right sub-tree, and then print the operator (`+`, `-`, `*`, or `/`, as appropriate). Each of these should be followed by a space. (**O**(*n*))

`bool is_leaf() const`
: Returns `true` if the node is a leaf noder. (**O**(1))

Team Wyoming

Lazy-deletion tree

Requirements:

In this project, you will create the classes:

1. Lazy-deletion binary search tree: `Lazy_deletion_tree`.
2. Lazy-deletion binary search node: `Lazy_deletion_node`.

A lazy-deletion binary search tree is a binary search tree where erased objects are simply tagged as erased while the nodes themselves remain in the tree. Occasionally, a member function may be called to *clean up* (delete) all erased nodes at once. Almost all functions will be implemented by calling the corresponding function on the root node.

Runtime:

The run time of each member function is specified in parentheses at the end of the description. The variable *n* is the number of nodes in the tree while the variable *h* is the height of the tree. Because this is not a balanced tree, a sequence of poor insertions may result in a tree of height **O**(*n*).

Class Specifications:

`Lazy_deletion_tree`

Description

A class which implements a lazy-deletion tree. Types which are erased from the tree are simply tagged as being erased.

Member Variables

This class two member variables:

- A pointer to the root node, and
- A variable storing the number of non-erased objects in the tree.

Accessors

This class has seven accessors:

`bool empty() const`

Return true if the tree is empty (the size is 0). (**O**(1))

`int size() const`

Returns the number of nodes in the tree not including nodes tagged as erased. (**O**(1))

`int height() const`

Returns the height of the tree including nodes tagged as erased. (**O**(*n*))

`bool member( Type const &obj ) const`

Return true if the argument is in the tree and not tagged as erased and false otherwise. (**O**(*h*))

`Type front() const`

Return the minimum non-erased object of this tree by calling `front` on the root node. The object will be the first argument of a pair `(*, true)` returned by the node member function, and will throw an exception `underflow` if the second for `(*, false)` (the tree has size zero). Hint: What type of traversal will you need? Under what conditions do you continue searching, and under what conditions do you return? (**O**(*n*))

`Type back() const`

Return the maximum non-erased object of this tree by calling `back` on the root node. The object returned will be a pair of the form `(*, true)`; return the first object. This member function may throw an `underflow` exception if the tree has zero size. (**O**(*n*))

`void breadth_first_traversal() const`

Perform a breadh-first traversal which prints the objects in the order in which they are visited in a single line (no end-of-line character). If an object is marked as erased, the string `"x "` is printed immediately following the object, otherwise a string containing a single space `" "` is printed after the object. You may use the Standard Template Library (STL) `std::queue` for the queue required for the traversal. For example, valid (ignoring the quotation marks) output may be `"3 7x 4x 9 5x "` (**O**(*n*))

Mutators

This class has four mutators [In computer science, a **mutator** method is a method used to control changes to a variable]:

`bool insert( Type const & )`

Insert the new object into the tree: If the object is already in the tree and not tagged as erased, return false and do nothing; if the object is already in the tree but tagged as erased, untag it and return true; if the object is not in the tree, create a new leaf node in the appropriate location and return true. If the root node is `nullptr`, this requires the creation of a new root node; otherwise, the corresponding function is called on the root node. (**O**(*h*))

`bool erase( Type const & )`

Removes the object from the tree: If the object is not already in the tree, return false; if the object is in the tree but tagged as erased, return false; if the object is in the tree and not tagged as erased, tag it as erased and return true. If the root node is `nullptr`, all that is done is that false is returned; otherwise, the corresponding function is called on the root node. (**O**(*h*))

`void clear()`

Delete all the nodes in the tree. (**O**(*n*))

`void clean()`

Delete all nodes tagged as deleted within the tree following the description found in the lazy-deletion node class. (**O**(*n*) if the height is **Θ**(ln(*n*)) but **O**(n^2) in general)

Development and Testing

Before you begin coding, consider the lazy-deletion trees in Figure 1 where some nodes have already been marked as deleted (grayed out). Your most difficult task will be to write the `clean()` member function. Make sure that you know conceptually what you would do before you start coding. If you don't understand what you're doing, you won't be able to code it either.

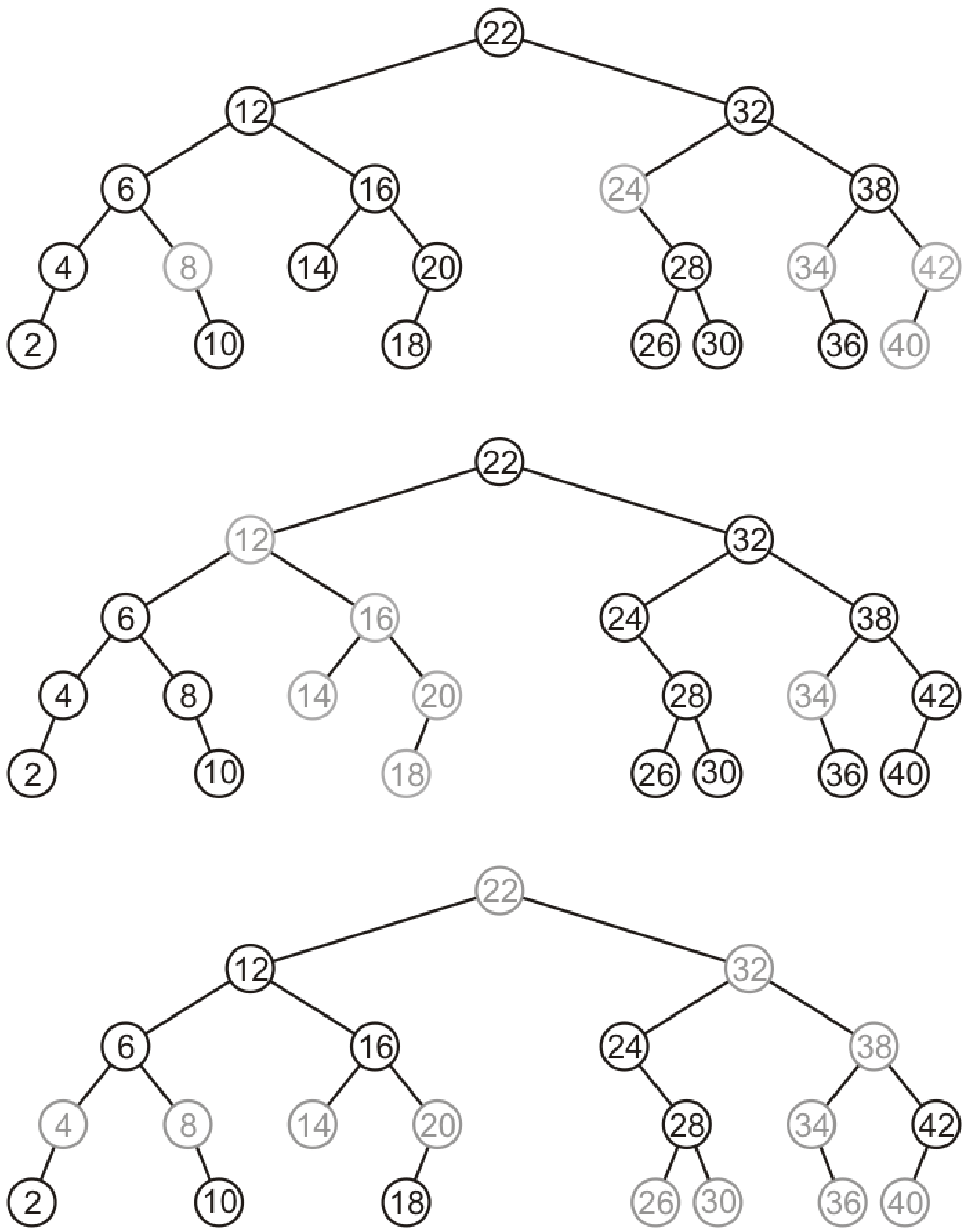

Figure 1. Various examples of lazy-deletion trees.

These are not meant to be exhaustive in enumerating all possible situations or cases.

Team Nebraska

Dynamic Linear Hash Table

Requirements:

In this project, you will implement one class:

1. Dynamic Linear-Probing Hash Table: Dynamic_linear_hash_table.

A hash table stores positive integers (≥ 0) in the bin corresponding to the hash or searches forward if that bin is filled.

The load factor of a hash table may increase array may be changed depending on the number of positive integers currently stored in the array, according to the following two rules:

- If a positive integer is being inserted into a hash table where the load factor after the insertion is 0.75, the size of the array is doubled.
- If, after removing a positive integer from a hash table where the load factor after the removal is 0.25, then the size of the array is halved. The size of the array may not be reduced below the initially specified size.

Runtime:

The run time of each member function is specified in parentheses at the end of the description.

Class Specifications:

`Dynamic_linear_hash_table`

Description

Create a hash table which uses linear probing where the array size may double or halve, depending on the load factor.

Member Variables

The class at least six members:

- A pointer to an `int`, `int *array`, to be used as an array of bins,
- An initial capacity `int initial_capacity`,
- A current capacity `int current_capacity`, and
- A counter `int count`.

Hash Function

The hash value of any positive integer will be:

- multply the number by 53267,
- if this result is negative, add 2^{31},
- ignore the five least-significant bits and use an appropriate modulo of the balance to get the number.

For example, to get either a 4-bit or a 7-bit hash value of 1, 53, 93, and 100, we have 0 and 0; 15 and 31; 7 and 55; and 11 and 59. For example, if the hash table is empty and of capacity 2^4, then 100 would be stored in bin 11, and if the hash table is empty and of capacity 2^7, then 100 would be stored in bin 59.

Member Functions

Constructors

```
Dynamic_linear_hash_table( int n = 3 )
```

The constructor takes as an argument the initial size of the array and allocates memory for that array. The default number of entries is 2^n. You may assume that $n \geq 2$ (that is, there will be at least 4 bins). Other class members are assigned as appropriate. (**O**(2^n))

Destructor

```
~Dynamic_linear_hash_table()
```

The destructor deletes the memory allocated for the array.

Accessors

This class has six accessors:

`int size() const`
: Returns the number of positive integers currently stored in the hash table. (**O**(1))

`bool empty() const`
: Returns `true` if the hash table is empty, `false` otherwise. (**O**(1))

`int capacity() const`
: Returns the current size of the array. (**O**(1))

`bool member( int ) const`

Returns true if the argument is in the hash table, and false otherwise. (**O**(1) on average)

`double load_factor() const`

Returns the load factor as a double-precision floating-point number. (**O**(1))

`int bin( int ) const`

Returns whatever is stored in the bin number corresponding to the argument (this will only be called on bins which I expect to be filled). (**O**(1))

Mutators

This class has three mutators [In computer science, a **mutator** method is a method used to control changes to a variable]:

`void insert( int )`

Insert the new integer into the hash table. Linear probing should be used if any of the bins are full. If the addition causes the load factor to reach 0.75, then a new hash table twice the size should be allocated and reinsert all the positive integers. The argument will **always** be positive. (**O**(1) on average)

`bool remove( int )`

Remove the integer from the hash table. If, after the deletion, the load factor drops to 0.25 or less, then a new hash table half the size should be allocated and reinsert all the positive integers (except, of course, the removed one). The size of array must never go below the initial size. Be sure to make sure that all entries can still be found. If the integer is not found, return false. (**O**(1) on average)

`void clear()`

Removes all the positive integers in the hash table. If the array is not the initial size, it, too, should be resized to the initial capacity. (**O**(2^n) where *n* is the initial argument to the constructor)

Team Michigan

Dckoo hash table

Requirements:

In this project, you will implement one class:

1. Cuckoo hash table: Cuckoo_hash_table.

A hash table stores positive integers (≥ 0) in the bin corresponding to the hash or searches forward if that bin is filled.

The load factor of a hash table may increase array may be changed depending on the number of positive integers currently stored in the array, according to the following two rules:

- If a positive integer is being inserted into a hash table where the load factor after the insertion is 0.75, the size of the array is doubled.
- If, after removing a positive integer from a hash table where the load factor after the removal is 0.25, then the size of the array is halved. The size of the array may not be reduced below the initially specified size.

Runtime:

The run time of each member function is specified in parentheses at the end of the description.

Class Specifications:

`Dynamic_linear_hash_table`

Description

Create a hash table which uses linear probing where the array size may double or halve, depending on the load factor.

Member Variables

The class at least six members:

- A pointer to an `int`, `int *array`, to be used as an array of bins,
- An initial capacity `int initial_capacity`,
- A current capacity `int current_capacity`, and
- A counter `int count`.

Hash Function

The hash value of any positive integer will be:

- multply the number by 53267,
- if this result is negative, add 2^{31},
- ignore the five least-significant bits and use an appropriate modulo of the balance to get the number.

For example, to get either a 4-bit or a 7-bit hash value of 1, 53, 93, and 100, we have 0 and 0; 15 and 31; 7 and 55; and 11 and 59. For example, if the hash table is empty and of capacity 2^4, then 100 would be stored in bin 11, and if the hash table is empty and of capacity 2^7, then 100 would be stored in bin 59.

Member Functions

Constructors

```
Dynamic_linear_hash_table( int n = 3 )
```

The constructor takes as an argument the initial size of the array and allocates memory for that array. The default number of entries is 2^n. You may assume that $n \geq 2$ (that is, there will be at least 4 bins). Other class members are assigned as appropriate. (**O**(2^n))

Destructor

```
~Dynamic_linear_hash_table()
```

The destructor deletes the memory allocated for the array.

Accessors

This class has six accessors:

```
int size() const
```

Returns the number of positive integers currently stored in the hash table. (**O**(1)) `bool empty() const`

Returns `true` if the hash table is empty, `false` otherwise. (**O**(1))

`int capacity() const`

Returns the current size of the array. (**O**(1))

`bool member( int ) const`

Returns true if the argument is in the hash table, and false otherwise. (**O**(1) on average)

`double load_factor() const`

Returns the load factor as a double-precision floating-point number. (**O**(1))

`int bin( int ) const`

Returns whatever is stored in the bin number corresponding to the argument (this will only be called on bins which I expect to be filled). (**O**(1))

Mutators

This class has three mutators [In computer science, a **mutator** method is a method used to control changes to a variable]:

`void insert( int )`

Insert the new integer into the hash table. Linear probing should be used if any of the bins are full. If the addition causes the load factor to reach 0.75, then a new hash table twice the size should be allocated and reinsert all the positive integers. The argument will **always** be positive. (**O**(1) on average)

`bool remove( int )`

Remove the integer from the hash table. If, after the deletion, the load factor drops to 0.25 or less, then a new hash table half the size should be allocated and reinsert all the positive integers (except, of course, the removed one). The size of array must never go below the initial size. Be sure to make sure that all entries can still be found. If the integer is not found, return false. (**O**(1) on average)

`void clear()`

Removes all the positive integers in the hash table. If the array is not the initial size, it, too, should be resized to the initial capacity. (**O**(2^n) where *n* is the initial argument to the constructor)

Team Minnesota

Double Hash Tables

A single testing file has been provided.

Requirements:

In this project, you will implement one class:

1. Hash Tables using Double Hashing: `Double_hash_table`.

In this project, you will create a hash table which stores objects. We will assume that the hash functions are appropriate so as to allow all operations to run in **Θ**(1) time when the load factor is not greater than, say, 0.666.

Run time:

The run time of each member function is specified in parentheses at the end of the description. We assume that the distribution of the hash function is even.

Class Specifications:

`Double_hash_table`

Description

A class which implements a hash table using double hashing. For run-time requirements, the number of elements in the hash table is *n* and the size of the hash table is *M*.

The primary hash function (that determining the bin) is the object statically cast as an `int` (see `static_cast<int>`) and taking this integer module *M* (`i % M`) and adding *M* if the value is negative. The secondary (odd) hash function (that determining the jump size) is **derived** from the integer divided by *M* modulo *M* (`(i / M) % M`) which is again made positive, if necessary, by adding *M*.

Member Variables

The class has at least three members:

- An array of objects `Type *array`,
- An array size `int array_size`, and
- A count `int count`.

You may need additional member variables.

Member Functions

Constructors

The constructor takes an argument `m` and creates a hash table with 2^m bins, indexed from 0 to $2^m - 1$. The default value of `m` is 5.

Destructor

The destructor frees up any memory allocated by the constructor.

Accessors

This class has seven accessors:

`int size() const`
: Returns the number of elements currently stored in the hash table. (**Θ**(1))

`int capacity() const`
: Returns the number of bins in the hash table. (**Θ**(1))

`double load_factor() const`
: Returns the load factor of hash table (see `static_cast<double>(...)`). This should be the ratio of occupied bins over the total number of bins. (**Θ**(1))

`bool empty() const`
: Returns `true` if the hash table is empty, `false` otherwise. (**Θ**(1))

`bool member( Type const & ) const`
: Returns `true` if object `obj` is in the hash table and `false` otherwise. (**Θ**(1) under our assumptions)

`Type bin( int n ) const`
: Return the entry in bin `n`. The behaviour of this function is undefined if the bin is not filled. It will only be used to test locations that are expected to be filled by specific values. (**Θ**(1))

`void print() const`
: A function which you can use to *print* the class in the testing environment. This function will not be tested.

Mutators

This class has three mutators [In computer science, a **mutator** method is a method used to control changes to a variable]:

`void insert( Type const & )`

Insert the argument into the hash table in the appropriate bin as determined by the two aforementioned hash functions and the rules of double hashing. If the table is full, thrown an `overflow` exception. If the argument is already in the hash table, do nothing. An object can be placed either into an empty or deleted bin. (**Θ**(1) under our assumptions)

`bool erase( Type const & )`

Remove the argument from the hash table if it is in the hash table (returning `false` if it is not) by setting the corresponding flag of the bin to deleted. (**Θ**(1) under our assumptions)

`void clear()`

Removes all the elements in the hash table. (**Θ**(M))

Team Utah

Dynamic Double Hash Tables

Requirements:

In this project, you will implement one class:

1. Dynamically Resizing Hash Tables using Double Hashing: Dynamic_double_hash_table.

In this project, you will create a hash table which stores objects. We will assume that the hash function is sufficiently even so as to allow all expected constant-time operations to be **O**(1).

Run time:

The run time of each member function is specified in parentheses at the end of the description. We assume that the distribution of the hash function is even.

Class Specifications:

`Dynamic_double_hash_table`

Description

A class which implements a hash table using double hashing. For run-time requirements, the number of elements in the hash table is *n* and the size of the hash table is *M*.

The primary hash function (the one determining the bin) is the object statically cast as an `int` and taking this integer modulo *M* (`i % M`) and adding *M* if the value is negative. The secondary (odd) hash function (the one determining the jump size) is **derived** from the integer divided by *M* modulo *M* (`(i / M) % M`) which is again made positive, if necessary, by adding *M*.

Member Variables

The class has at least five members:

- An array of objects `Type *array`,
- An array storing the current state of a bin `state *occupied`,
- An array size `int array_size`, and

- A count `int count`.
- A count of deleted bins `int deleted_count`.

You may need additional member variables. An enumerated type `state` is provided; however, you may use a different solution.

Member Functions

Constructors

The constructor takes an argument `m` and creates a hash table with 2^m bins, indexed from 0 to $2^m - 1$. The default value of `m` is 5 and if $m < 2$, use the value $m = 2$, i.e., the initial size is no smaller than 4.

Destructor

The destructor frees up any memory allocated by the constructor.

Accessors

This class has seven accessors:

`int size() const`
    Returns the number of elements currently stored in the hash table. (**O**(1))
`int capacity() const`
    Returns the number of bins in the hash table. (**O**(1))
`double load_factor() const`
    Returns the load factor of hash table (see `static_cast<double>(...)`). (**O**(1))
`double deleted_factor() const`
    Returns the number of deleted bins over the number of bins. (**O**(1))
`bool empty() const`
    Returns `true` if the hash table is empty, `false` otherwise. (**O**(1))
`bool member( Type const & ) const`
    Returns `true` if object `obj` is in the hash table and `false` otherwise. (Amortized **O**(1))
`Type bin( int ) const`
    Return the entry in bin n. The behaviour of this function is undefined if the bin is not filled. It will only be used to test the class with the expected locations. (**O**(1))

Mutators

This class has three mutators [In computer science, a **mutator** method is a method used to control changes to a variable]:

`void insert( Type const & )`
    If the object is currently a member of the hash table, do not add it. Insert a new object into the hash table in the appropriate bin as determined by the two aforementioned hash

functions and the rules of double hashing. If before the insertion the load factor greater or equal than 0.75, double the size of the hash table and rehash all original entries into the new hash table in the order in which they appear in the current hash table prior to doubling. Only add the new entry after any doubling has occured. (Amortized **O**(1))

`bool remove( Type const & )`

Remove the object from the hash table if it is in the hash table (returning `false` if it is not) by setting the corresponding flag of the bin to deleted. If the argument was successfully removed and the load factor is lesser or equal than 0.25 and the hash table is not already the initial size of the hash table, halve the size of the hash table and insert the entries into the new hash table in the order in which they appear in the current hash table prior to halving. If the table was not halved in size but the proportion of deleted bins is greater or equal than 0.25, rehash all the entries into the same sized hash table in the order(ordered from index 0 to M) in which they appear in the current hash table. (Amortized **O**(1))

`void clear()`

Removes all the elements in the hash table and resets the size to the initial size. (**O**(M))

It is suggested that you have one function which deals with halving, rehashing, and doubling the hash table.