ამ დავალებაში თქვენი ამოცანაა შექმნათ პრიორიტეტული რიგის კოლექცია კლასი. პრიორიტეტული რიგი არის სტანდარტული რიგის შეცვლილი ვარიანტი და ელემენტების ამოღება(dequeue) რიგითობის ნაცვლად ხდება პრიორიტეტის მიხედვით.   
მაგალითისთვის ასეთი კოლექცია გამოსადეგი იქნება საავადმყოფოს რიგის მოდელიირების დროს. რიგში მყოფ ყველა პაციენტს ექნება პრიორიტეტი, თუ რამდენად კრიტიკულია მისი მდგომარეობა, შესაბამისად პრიორიტეტული რიგი, ყოველ ჯერზე მოგვცემს პაციენტის ნომერს რომლის მდგომარეობაც ყველა სხვა პაციენტზე უფრო მძიმეა.  
დავალების მთავარ ფოკუსს წარმოადგენს priority queue-ს იმპლემენტაცია რამდენიმე განსხვავებული გზით. გექნებათ შანსი ექსპერიმენტების აწარმოოთ მასივზე, ბმულ სიებზე და სპეციალური სახის ხეზე, heap -ზე. როდესაც საკუთარი კოდის დებაგინგს დაასრულებთ, საშუალება გექნებათ სხვადასხვა ტესტებით შეაფასოთ სხვადასხვანაირი იმპლემენტაციის ძლიერი და სუსტი მხარეები.

**პრიორიტეტული რიგის არსი**

ამ დავალებაში თქვენ დაწერთ პრიორიტეტული რიგის რეალიზაციას, რომელიც ინახავს სტრინგებს. ელემენტის ამოღებისას(dequeue) პრიორიტეტულმა რიგმა უნდა წაშალოს და დააბრუნოს ლექსიკოგრაფიულად პირველი სტრინგი.

მაგალითისთის თუ თქვენ ჩაამატებთ პრიორიტეტულ რიგში სტრინგების:  
 “Goldilocks,” “Mama bear,” “Papa bear,” “Baby bear”

ამოღებსას რიგმა უნდა დააბრუნოს შემდეგი მიმდევრობით:  
 “Baby bear,” “Goldilocks,” “Mama bear,” “Papa bear”

პრიორიტეტული რიგის ინტერფეისი შედგება შემდეგი ოპერაციებისგან:

* size - აბრუნებს რიგში ელემენტების რაოდენობას
* isEmpty - აბრუნებს true\false იმის მიხედვით ცარიელია თუ არა რიგი
* enqueue - რიგში ელემენტის ჩამატება
* peek - აბრუნებს, მაგრამ არ შლის ლექსიკოგრაფიულად პირველ ელემენტს. თუ რიგი ცარიელია, უნდა მოხდეს error ფუნქციის გამოძახება.
* dequeueMin - აბრუნებს და შლის ლექსიკოგრაფიულად პირველ ელემენტს. თუ რიგი ცარიელია, უნდა მოხდეს error ფუნქციის გამოძახება

**დავალება**

როგორც ლექციებზე ნახეთ ერთი და იგივე კოლექციის კლასი შეიძლება რამდენიმე გზით დაიწეროს. თქვენი დავალებაა დაწეროთ პრიორიტეტული რიგის რეალიზაცია დაწეროთ ოთხი სხვადასხავ გზით:

* **დაულაგებელი ვექტორი.** ელემენტები ინახება ვექტორში არეული მიმდევრობით.
* **დალაგებილი ბმული სია.** ელემენტები ინახება დალაგებული სახით ცალმხრივი ბმული სიის საშუალებით.
* **დაულაგებელი ბმული სია.** ელემენტები ინახება ორმხრივი ბმული სიის საშუალებით.
* **ორობითი ჰიპი.** ელემენტები ინახება სპეციალურ სტრუქტურაში - *binary heap*.

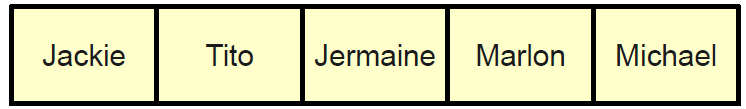
ჩვენი რეკომენდაცია იქნება, რეალიზაციები დაწეროთ ზემოთ ჩამოთვლილი მიმდევრობით.

**რეალიზაცია 1 - რეალიზაცია დაუსორტირებელი ვექტორით**

დაულაგებელი ვექტორით იმპპპლემენტაცია ყველაზე მარტივი შემთხვევაა და ემსახურება კლასის სტრუქტურასთან გაცნობას.  
pqueue-vector.h ფაილში ნახავთ VectorPriorityQueue კლასის ინტერფეისს. თქვენი ამოცანაა ჰედერ ფაილში არსებული ფუნქციების რეალიზაციის დაწერა. ამისათვის დაგჭირდებათ private ველების შემოტანა და pqueue-vector.cpp ფაილში რეალიზაციის კოდის დაწერა.

რიგში ახალი ელემენტის დამატებისას თქვენ უბრალოდ უნდა ჩაამატოთ ეს ელემენტი რიგის შიგნით არსებულ ვექტორში. მაგალითისთის თუ თქვენ ჩაამატებთ პრიორიტეტულ რიგში სტრინგების:  
 “Jackie”, “Tito”, “Jermaine”, “Marlon”, “Michael”

რიგის შიგნით განმარტებული ვექტორი შემდეგ სახეს მიიღებს:



რიგიდან სტრინგის ამოღებისას, საჭიროა ვექტორში ლექსიკოგრაფიულად მინიმალური სიტყვის პოვნა, ვექტორიდან წაშლა და დაბრუნება. მაგალითად ზემოთ ნაჩვენები რიგისთვის dequeueMin დააბრუნებს “Jackie” -ის ხოლო რიგი მიიღებს შემდეგ სახეს:



pqueue-test.cpp ფაილში მოცემულია კოდი რომლის საშუალებითაც შეგიძილიათ თქვენი პრიორიტეტული რიგი შეამოწმოთ სხვადასხვა ტესტებზე.

VectorPriorityQueue-ის რეალიზაციის დაწერის შემდეგ დაფიქრდით რამდენიმე საკითხზე:

* ელემენტის ჩამატებისას რა რაოდენობის( big-O) მოქმედებები ჭირდება რიგს, საუკეთესო შემთხვევაში, ყველაზე ცუდი შემთვხვევაში.
* ელემენტის ამოღებისას რა რაოდენობის( big-O) მოქმედებები ჭირდება რიგს, საუკეთესო შემთხვევაში, ყველაზე ცუდი შემთვხვევაში.
* თუ რიგში ჩაამატებთ n ცალ ელემენტს და შემდეგ ამოიღეთ მათ რიგრიგობით, მიიღებთ ელემენტებს დალაგებული სახით. რომელი სორტირების ალგორითმს გავს ასეთი სახით დალაგება?

**რეალიზაცია 2 - დალაგებული ბმილი სია**

დაულაგებელი ვექტორის შემთხვევაში, ელემენტის ჩამატება სწრაფად ხდება, სამაგიეროდ ამოღება შრომატევადი ოპერაციაა. ალტერნატიული გზა არის ელემენტების დალაგებული სახით შენახვა, რაც აასწრაფებს ელემენტის ამოღებას, მაგრამ ჩამატება გართულდება, რადგან ყოველი ახალი ელემენტი უნდა ჩაემატოს ისეთ ადგილზე რომ სია დასორტირებული დარჩეს.

ამ ჯერად თქვენი ამოცანა იქნება დაწეროთ პრიორიტეტული რიგის რეალიზაცია ცალმხრივი დალაგებული ბმული სიის საშუალებით. ანუ რიგში ჩამატებული ელემენტები უნდა შეინახოთ ბმული სიის საშუალებით, ისე რომ სია ყოველთვის დასორტირებული იყოს.   
pqueue-linkedlist.h ჰედერ ფაილში გაწერილია LinkedListPriorityQueue კლასის ინტერფეისი, თქვენ უნდა შემოიტანოთ საჭირო ველები და დაწეროთ ამ კლასის რეალიზაცია pqueue-linkedlist.cpp ფაილში.

მაგალითისთვის დავუშვათ რიგში ჩავამატეთ შემდეგი სტრინგები:

“Primrose Everdeen,” “Katniss Everdeen,” “Peeta Mellark”

ასეთ შემთხვევაში რიგი მიიღებს შემდეგ სახეს:



როგორც ხედავთ ბმულ სიაში ელემენტები დალაგებულია ლექსიკოგრაფიულად ზრდადობის მიხედვით.   
თუ რიგში ჩავამატებთ სტრინგს “Gale Hawthorne”, სიტუაცია შეიცვლება შემდეგნაირად:



ხოლო “Haymitch Abernathy,”, ის ჩამატების შემთხვევაში მივიღებთ შემდეგ სურათს:



ელემენტების ამოთება ტრივიალურია, რადგან სია დალაგებულია და ყოველთვის ბმული სიის პირველი ელემენტი უნდა დაბრუნდეს.

რეალიზაციის ყველაზე რთული მომენტი არის ახალი ელემენტის ჩამატება, რადგან საჭიროა ბმულ სიაში სპეციალური ადგილის პოვნა და შემდეგ ახალი წევრის ჩამატება, ისე რომ სია დალაგებული დარჩეს.

LinkedListPriorityQueue ის რეალიზების შემდეგ დაფიქრდით რამდენიმე საკითხზე:

* ელემენტის ჩამატებისას რა რაოდენობის( big-O) მოქმედებები ჭირდება რიგს, საუკეთესო შემთხვევაში, ყველაზე ცუდი შემთვხვევაში.
* ელემენტის ამოღებისას რა რაოდენობის( big-O) მოქმედებები ჭირდება რიგს, საუკეთესო შემთხვევაში, ყველაზე ცუდი შემთვხვევაში.
* თუ რიგში ჩაამატებთ n ცალ ელემენტს და შემდეგ ამოიღეთ მათ რიგრიგობით, მიიღებთ ელემენტებს დალაგებული სახით. რომელი სორტირების ალგორითმს გავს ასეთი სახით დალაგება?

**რეალიზაცია 3 - დუალაგებული ორმაგი ბმილი სია**

ბმული სიის ალტერნატიული ვარიანტი არის ორმაგი ბმული სია, სადაც ყოველ ელემენტი ინახავს სიაში მომდევნო და წინა წევრების მისამართებს, რაც გვაძლევს ორმხრივი ნავიგაციის საშუალებას.

ამ ნაწილში თქვენი დავალება იქნება პრიორიტეტული რიგის იმპლემენტაცია დაულაგებელი ორმხრივი ბმული სიის საშუალებით. რიგში ელემენტის ჩამატებისას, ახალი ელემენტი უნდა ჩაამატოთ ბმული სიის თავში. მაგალითად თუ რიგში ვამატებთ ელემენტებს:

“Marten Reed,” “Faye Whitaker,” “Dora Bianchi”

ბმულ სიაში ელემენტები შემდეგი სახით იქნება დალაგებული:



ელემენტის ამოღებისას უნდა გადაუაროთ სიას, იპოვოთ მინიმალური, წაშალოთ ბმული სიიდან და დააბრუნოთ.  
pqueue-doublylinkedlist.h ჰედერ ფაილში გაწერილია კლასის ინტერფეისი, თქვენ უნდა შემოიტანოთ private ველები და დაწეროთ თითოეული ფუნქციის იმპლემენტაცია pqueue-doublylinkedlist.hფაილში.

**რეალიზაცია 3 - ჰიპი**

ბოლო რეალიზაციაში ელემენტების შენახვისთვის უნდა გამოიყენოთ Binary Heap ის სტრუქტურა(იხ. სურათი).



ჰიპში ელემენტების მნიშნელობები ინახება *node* ებში, თითოეულ node-ს შეიძლება ყავდეს ერთი, ორი ან არცერთი შვილი, მაგალითად A ს ყავს ორი შვილი C და B, ხოლო E ს ყავს მხოლოდ ერთი შვილი F.   
ჰიპში ყველა ჰორიზონტალური ხაზი გარდა ბოლო ხაზისა მთლიანად უნდა იყოს შევსებული. ანუ პირველ ხაზში უნდა იყოს ერთი node, მეორე ხაზში ორი, მესამეში 4, მეოთხეში 8 და ასე შემდეგ, ბოლო ხაზი შეიძლება იყოს ნაწილობრივ შევსებული. მაგალითისთვის იხილეთ კიდევ ორი ხე:



სტრუქტურა ჰიპს აქვს კიდევ ერთი წესი, შვილობილი node არასდროს არ არის მშობელზე ლექსიკოგრაფიულად წინ. როგორც ნახეთ სამივე მაგალითში ეს წესი დაცული, თუმცა ელემენტების განლაგება შეიძლება სხვადასხვანაირი იყოს.

როცა პრიორიტეტული რიგი ელემენტებს ინახავს ჰიპში, მინიმალურის პოვნა ძალიან მარტივია, რადგან წესების თანახმად ყოველთვის root ელემნეტი იქნება მინიმალური. ასევე ეფექტურად შეიძლება გაკეთდეს ახალი ელემენტის ჩამატება.



სურათზე ნაჩვენებ ჰიპში ჩვამატოთ სიტყვა ”Gil“, პირველი წესის თანახმად ერთადერთი ადგილი სადაც სიტყვა შეიძლება ჩავამატოთ არის ბოლო ხაზი, ანუ “Lucrezia” ს მარცხენა შვილად.



თუმცა ასეთი სახით ჩენ ვარღვევთ მეორე წესს, რადგან Gil ლექსიკოგრაფიულად უფრო წინ დგას ვიდრე Lucrezia. ამ პრობლემის მოსაგვარებლად გამოვიყენოთ bubble-up მეთოდი, Gil ს გავუცვალოთ ადგილი მის მშობელთან მანამ სანამ საჭირო ადგილზე არ მოხვდება.



გაცვლის შემდეგ მივიღეთ ვალიდური heap. შემდეგი ჩავამატოთ სიტყვა “Dpree”, პირველად სიტყვას ვამატებთ Gil ის მარჯვენა შვილად ბოლო სტრიქონში.



შემდეგ ვიწყებთ გაცვლის პროცესს და სურათი იცვლება შემდეგანაირად:



ბოლო მაგალითისთვის ჩავამატოთ სიტყვა “Othar”, თავდაპირველად სიტყვას ვამატებთ პირველ თავისუფალ ადგილზე, რადგან ამ შემთხვევაში ბოლო სტრიქონი მთლიანად შევსებულია, ვიწყებთ ახალი სტრიქონიდან და სიტყვას ვამატებთ Zeetha ს მარცხენა შვილად.



bubble-up პროცესის საშუალებით მოხდება ორი გაცვლა, ჯერ Zeetha და Othar ს შორის და შემდეგ Tarvek -სა და Othar ს შორის, შესაბამისად მივიღებთ ახალ სურათს:



მარტივად შეიძლება დათვლა რომ ჩამატება O(log n) დროში ხდება, რადგან n ელემენტისგან შედგენილი ხის სიღრმე log n იქნება.

ამგვარად ელემენტის ჩამატება სრულიად ნათელია, გვერჩემა ამოღებუს მეთოდის რეალიზაცია. ცხადია რომ ჰიპის პირველი ელემენტი არის მინიმალური, მაგრამ მისი წაშლით ჩენ ხეს ორ ჰიპად გავხლიჩავთ. ამიტომ მივმართოთ სხვა გზას, პირველ რიგში ჰიპის პირველ ელემენტს გავუცვალოთ ადგილი ბოლო ელემენტთან:



ახლა მარტივად შეგვიძლია ხიდან წავშალოთ Agatha:



სამწუხაროდ მიღებული ხე არ აკმაყოფილებს ფიპის წესებს, რადგან Zeetha უფრო უკან დგას ლექსიკოგრაფიულად ვიდრე მისი შვილები. ამ პრობლემის გამოსასწორებლად მივმართოთ bubble-down მეთოდს, გავუცვალოთ Zeetha ადგილი უმცურეს შვილთან სანამ ეს შესაძლებელია. პირველად ადგილებს ვუცვლით Zeetha ს და Dupree ს, შემდეგ Zeetha ს და Gil ს:



სულ ეს არის, მივიღეთ ჰიპი. როგორც ვხედავთ ელემენტის ჩამატებაც და ამოღებაც შესაძლებელია O(log n) დროში, რაც ძალიან კარგი შედეგია. N ელემენტის ჩამატება და შემდეგ რიგიდან ამოღებას დაჭირდება O (n log n) დრო, ასეთი სორტირების მეთოდს heapsort ეწოდება.

ჰიპის წარმოდგენა რამდენიმე სახით შეიძლება, თუმცა ამ დავალებაში გამოვიყენებთ უბრალოდ დინამიურ მასივს. მასივის სახით შენხვის დროს, მთავარ როლს თამაშობს ელემენტების გადანომრვა, მაგალითად ზემოთ ნაჩვენებ ჰიპი გადაინომრება შემდეგი სახით:



ნუმერაციის მეთოდი ძალიან მარტივია:

* თუ node ის ნომერი არის n მაშინ მისი შვილების ნომრები იქნება 2n და 2n + 1
* თუ node ის ნომერი არის n მაშინ მისი მშობლის ნომერი იქნება n/2

ამ წესებით ზემოთ ნაჩვენები ჰიპი, მასივის სახით შემდეგნაირად გამოიყურება:  


ელემენტის ჩამატების და ამოღების ალგორითმები მარტივად რეალიზაებადია მასივის შემთვევაშიც. როგორც ითქვა ახალი ელემენტის დამატებისას, ჯერ ვამატებთ მას ხის ბოლოში, ეს კი იგივეა რაც მასივის ბოლო ელემენტად დამატება. ამის შემდეგ საჭიროა მშობელთან ადგილის გაცვლა, რაც ასე ვმარტივად მოხდება რადგან მასივში მშობლის ინდექსის გამოთვლა ელემენტარულია.

ბოლო დავალებისთვის თქვენ უნდა დაწეროთ HeapPriorityQueue კლასის რეალიზაცია pqueue-heap.h და pqueue-heap.cpp ფაილებში.

**შენიშვნა:** გადანომრვისას ჩვენ გამოვიყენეთ ნუმერაცია 1 დან, გაითვალისწინეთ რომ C++ ში მასივის ელემენტები ინომრება 0 დან.

**Extra Credit Opportunity: Build Your Own Priority Queue!**

The four heap implementation strategies you will implement in this assignment are only a small sampling of the myriad implementations of priority queues. For extra credit, research one of the following priority queue implementations, then implement the ExtraPriorityQueue class using one of these more complex (but more efficient!) designs:

• Binomial heap: A type of heap built out of a collection of smaller trees called binomial trees. Binomial heaps support the merge operation efficiently: given two binomial heaps, it is possible to combine them together into a single binomial heap in time O(log n), where n is the total number of nodes in the two heaps.

• Fibonacci heap: If you're up for a real challenge, try implementing the Fibonacci heap! This data structure is a modified version of a binomial heap that supports the decrease-key operation efficiently. Instead of sorting the strings alphabetically, instead you would associate each string with a priority and then sort the strings by priority. The decrease-key operation then lets you low the priority of an already-enqueued element. Fibonacci heaps have excellent theoretical properties, but tend to be slow in practice.

• Pairing heap: A pairing heap is a simpler alternative to the Fibonacci heap that is slightly slower on huge inputs but is faster on small inputs.

**Advice, Tips, and Tricks**

Here are a few pointers that might make your life easier as you work through this assignment:

• Draw pictures. When manipulating linked lists, it is often helpful to draw pictures of the linked list before, during, and after each of the operations you perform on it. Manipulating linked lists can be tricky, but if you have a picture in front of you as you're coding it can make your job substantially easier.

• Don't panic! You will be doing a lot of pointer gymnastics in the course of this assignment, and you will almost certainly encounter a crash in the course of writing your program. If your program crashes, resist the urge to immediately make changes to your code. Instead, look over your code methodically. Use the debugger to step through the code one piece at a time, or use the provided testing harness to execute specific commands on the priority queue. The bug is waiting there to be found, and with persistence you will find it. If your program crashes with a specific error message, try to figure out exactly what that message means. Don't hesitate to get in touch with your section leader, and feel free to stop by the LaIR or office hours.

• Test thoroughly. We have provided you a fairly comprehensive testing system that you can use to verify your code. When you run the provided starter code, you will have the option to manually or automatically test all four of the priority queue implementations (plus the optional extra fifth). The manual tests are good for initial debugging; they allow you to directly issue commands to the priority queue and see what happens. Once you're comfortable that your implementation is mostly correct, you can run the automated tests. The automated tests will subject your priority queue to a battery of tests that will cover a lot of cases. We cannot guarantee that our automatic tests will cover every case, though, and you're strongly encouraged to add your own testing code.

Good luck!