CSE221 Data Structures (Spring 2018)
Instructor: Prof. Tsz-Chiu Au
Due date: May 15, 2018, 11:59 pm.

Assignment 3: Heap-based Priority Queues and Hash Maps

Priority queues and associative arrays are among the most common data structures in computer science. Some programming languages natively support these data structures even at the language level as well as in their built-in libraries. The strong support from programming languages make these data structures very easy to use—a programmer does not need to know much about their actual implementation before using them. However, the ignorance of the actual implementation of these data structures can cause inefficiency in the programs, especially when the performance of your programs depends on the running time of these data structures. In fact, a good programmer should possess the knowledge about the pros and cons of different implementations of these data structures in order to choose the right implementation for their applications.

In this exercise, you will gain the hand-on experience of two implementations of priority queues and associative arrays: heap-based priority queues and hash maps. Priority queues are often implemented by binary heaps, which offers O(log n) running time in most operations. Moreover, there is an array implementation of heaps that are space efficient. However, a proper implementation of heaps is not trivial—there are several tricks to make it works in practice. You will implement a priority queue with the following features:

- Array-based complete binary trees as the underlying implementation of the heap.
- The doubling strategy for increasing the size of the array when the heap is full.
- The support for the deletion of *any* element that is not the minimum element in the priority queue.
- Location-aware entries such that the position object always refers to the same element in the priority queue.

You will also implement a hash map with separate chaining as described in Section 9.2.7 in the textbook (Goodrich et al. (2011)). The implementation will be mostly the same as the textbook with two minor modifications that we will discuss later.

To illustrate the usage of the priority queue and the associative arrays, we provided you the code of a discrete-event simulation that utilizes both data structures. The simulation is about the assignment of jobs to machines. For example, imagine a machine is a printer, and a job is the print job that is submitted by a user. Suppose there are identical N printers, each of them can work on at most one print job at a time. If a job is assigned to a printer, the printer has to spend a certain amount of time *d* to finish a print job, where *d* is called the duration of the job. The users can submit job requests at any time, and they can also remove job requests that has been submitted previously. The job requests are stored in a job queue temporally and wait for a job scheduler to assign the jobs to the printer. Suppose the scheduler of the jobs assigns the print jobs to the printers in the *first-come, first-served* manner—whenever there is a free printer, it assigns the earliest unassigned print job in the job queue to a printer. The simulator can show what the printers do at different times.

The heart of the discrete-event simulation is a priority queue, which keeps track of the future events according to the ascending order of their time stamps. Notice that you are not asked to implement the discrete-event simulation—we have already provided you the code in `Simulator.h` and `Simulator.cpp`. However, some basic understanding of the code will be helpful to you.

Task 1: Implementing array-based complete binary trees with location-aware entries

The array-based complete binary tree is called the vector-based binary tree representation in the textbook. Section 8.3 in the textbook defines a class called `VectorCompleteTree`, which is based on `vector` in STL. However, we want you to implement the class using raw array (i.e., the built-in array in C++) instead, so that you can control the array directly. We will call the class that you will implement `ArrayCompleteTree`. In addition, you will also implement the *location-aware entries* as described in Section 8.4.2, such that the position object always points to the element no matter how the entry containing the element moves around in the data structure.

In `ArrayCompleteTree.h`, we have already included a skeleton of the `ArrayCompleteTree` class with all private variables you need as well as the inner classes for location-aware entries. You must use the skeleton to implement the complete tree data structure. More importantly,

(1) You are not allowed to define any other member variables (public, private, or protected) in `ArrayCompleteTree`, `ArrayCompleteTree::Locator`, and `ArrayCompleteTree::LocationAwareEntry`;
(2) You are not allowed to include any other header files except `iostream` and `stdexcept` that have already been included; and
(3) you are not allowed to define any public member functions or other classes in `ArrayCompleteTree.h`.

However, you are allowed to define any *private* member functions if you see fit.

Let us talk about the member variables first. In `ArrayCompleteTree`, there are three private member variables:

- `LocationAwareEntry* v`
- `int vsize`
- `int n`

`v` is an array for storing the location-aware entries and its size is `vsize`. When you resize `v` using the doubling strategy, remember to update `vsize`. `n` is the number of entries that has been used in `v`. Notice that `n` is always less than or equal to `vsize`. In the textbook, the first entry in `v` is left empty in order to save a few comparison operations in other member functions. However, we want you eliminate this empty entry to save some space instead. In the lecture, we have already discussed how to adjust the index calculations to avoid having an empty first entry. In our grading script, we will check whether you have removed this empty first entry.

The entries in `v` are location-aware entries, each of them has two member variables:

- `E e`
- `Locator *loc`

`e` is the place holder for the element you want to store in the entry. `loc` is a pointer to a `Locator` object, which stores the index `i` of the entry in `v`. When you copy or move your entry in `v`, please remember to update the index `i` in the Locator object so that it always refers to the correct index of the entry in `v`. Likewise, when you add or delete an entry in `v`, please remember to create or delete the `Locator` object.

The reason why we need to associate a `Locator` object to an entry is that when we set the `Position` object of an element to be a pointer to the `Locator` object, the `Position` object can always refer to the correct index of the element in `v`. Although the index of the entry storing the element can change from time to time, the address to the `Locator` object never changes. Hence, the `Position` objects that are created before the reorganization of entries in `v` will remain valid after the reorganization, since their `loc` always refer to the same `Locator` objects. The `Position` class has two member variables:

- `const ArrayCompleteTree* tree`
- `Locator *loc`

`loc` is the pointer to a `Locator` object. `tree` is a pointer to `ArrayCompleteTree.` You need `tree` in `Position` in order to support `operator*()`.

These are all the member variables you need, and you are not allowed to declare any other member variables in these classes. Moreover, all of these member variables should be private to the classes. In order to give `ArrayCompleteTree` the access to all private member variables, you should add the statement "`friend class ArrayCompleteTree<E>;`" in `Locator`, `LocationAwareEntry`, and `Position`.

Before we dive into the implementation of the member functions, let's talk about the set of all required public member functions that your class must support. You will have to implement the following public member functions in order to be accepted by our grading script.

- `ArrayCompleteTree`'s public constructors:
  - `ArrayCompleteTree(int _vsize = 10)`
  - `ArrayCompleteTree(const ArrayCompleteTree& t)`
- `ArrayCompleteTree`'s public destructor:
  - `~ArrayCompleteTree()`
- `ArrayCompleteTree`'s public member functions:
  - `int size() const`
  - `Position left(const Position& p) const`
  - `Position right(const Position& p) const`
  - `Position parent(const Position& p) const`
  - `bool hasLeft(const Position& p) const`
  - `bool hasRight(const Position& p) const`
  - `bool isRoot(const Position& p) const`
  - `Position root() const`
  - `Position last() const`
  - `void addLast(const E& e)`
  - `void removeLast()`
  - `void swap(const Position& p, const Position& q)`
  - `void cleanup()`
- `ArrayCompleteTree<E>::Locator`'s public constructor:
  - `Locator(int _i)`
- `ArrayCompleteTree<E>::LocationAwareEntry`'s public constructor:
  - `LocationAwareEntry()`
  - `LocationAwareEntry(E _e, int i)`
- `ArrayCompleteTree<E>::LocationAwareEntry`'s public member functions:
  - `void clear()`
- `ArrayCompleteTree<E>::Position`'s public constructor:

  - `Position()`
  - `Position(const ArrayCompleteTree *_tree, Locator *_loc)`
  - `bool operator==(const Position& p) const;`

- `ArrayCompleteTree<E>::Position`'s public member functions:

  - `E& operator*() const`

Most of the above member functions have been discussed in Section 8.3 of the textbook, though the implementation will be different. In the following, we will only talk about the member functions that are not discussed in the textbook.

`ArrayCompleteTree` should have two public constructors. The first one is the counterpart of `VectorCompleteTree::VectorCompleteTree()` in the textbook. However, we want you to set the default size of `v` to 10. The second constructor is a copy constructor. What it does is to make a copy of an `ArrayCompleteTree` object. Notice that you will have to duplicate `v` and the `Locator` objects in the copy constructor. Similarly, the destructor of `ArrayCompleteTree` frees the memory used by `v` as well as the `Locator` objects.

Most of the public member functions of `ArrayCompleteTree` have been described in Section 8.3.2 of the textbook. Please refer to the Code Fragment 8.11 on Page 342 for more detail. Notice that you should add `const` to the end of the declaration of `left()`, `right()`, and `parent()`. Once again, please make sure your code considers all cases and throw exceptions when appropriate.

You should pay attention to `addLast()`, which should double the size of `v` when `v` is full. Please refer to the lecture note for the doubling strategy. Remember to free the memory of the previous `v` and the `Locator` objects (once again, via the destructor of `LocationAwareEntry`) after resizing.

We would like you to implement a member function called `cleanup()`, which frees up the unused entries in `v` by shrinking the size of `v` down to `n`, the current number of elements in the tree. In other words, what `cleanup()` does is just like the doubling strategy in `addLast()`, except that the size of `v` is reduced to `n` instead. As you know, the doubling strategy can waste a lot of memory. When a program using `ArrayCompleteTree` does not need to add more element to the tree, it can call `cleanup()` to reduce the memory footprint of `ArrayCompleteTree` objects.

`Locator` is a very simple class and it stores only one integer, which is an index of `v`. However, you should be careful when implementing `LocationAwareEntry` in order to avoid corrupting the memory. Since `LocationAwareEntry` objects are entries in an array, there are certain constraints about how we can create and destroy them. One constraint is that it must have a default constructor that takes no argument (i.e., LocationAwareEntry()). However, you will need to explicitly declare this default constructor because the default constructor will not be automatically generated by the compiler when there is another constructor. Another constructor is LocationAwareEntry(E _e, int i), which initializes all member variables in an LocationAwareEntry object, including the creation of the Locator object.

You may be tempted to implement the destructor ~LocationAwareEntry(), which automatically deletes the Locator object when you delete v. However, when you implement the doubling strategy, you will find that ArrayCompleteTree needs to control when to create and delete Locator objects. In other words, you do not want the destructor to automatically delete of Locator objects for you. Instead, you should implement a member function called clear() in LocationAwareEntry, which deletes the Locator object and set loc to NULL. Then ArrayCompleteTree can decide when to delete the Locator objects using this function.

Position is also a very simple class. You need to explicitly declare the default constructor since some data structures (more specially, HashMap and Simulator::JobStatus) that uses the Position object need it. In general, all Position classes should have a default constructor for the same reason. operator*() returns a non-const reference to the element via the Locator object. It means that we can update the element referred by this Position object via this reference. Notice that while such assignment should be allowed in ArrayCompleteTree, HeapPriorityQueue should not use this reference to update or swap elements as this can violate the heap property. Apart from operator*(), you will need to implement operator==(), which compares two position objects to see whether both of them refer to the same entry in the same ArrayCompleteTree object, or both of them refer to nothing. There is a chance you will need to compare two position objects in HeapPriorityQueue<E,C>::remove().

Task 2: Implementing priority queues with the support of element removal.

Next, you will implement a priority queue called HeapPriorityQueue using ArrayCompleteTree. Section 8.3.4 in the textbook describes an implementation based on VectorCompleteTree. It is easy to replace VectorCompleteTree with ArrayCompleteTree since ArrayCompleteTree has the same interface as VectorCompleteTree. In fact, your code should be quite similar to the codes presented in Section 8.3.4. However, there are two differences that you should bear in mind:

- Your HeapPriorityQueue supports the removal of any element apart from the minimum element. The textbook calls such priority queues *adaptable priority queues*, but it seems that this is not a standard name for priority queues with element removal. Section 8.4 of the textbook presents a way to implement such priority queues, but the underlying data structure of the priority queue is not a heap but a list. In this exercise, we want you implement `remove()` for binary heaps. An excellent description of the removal algorithm is presented in http://www.mathcs.emory.edu/~cheung/Courses/171/Syllabus/9-BinTree/heap-delete.html. We will describe the algorithm below.

- Since the entries in `ArrayCompleteTree` are location-aware entries, you should be careful when swapping the elements in `insert()`, `removeMin()`, `remove()`, and `replace()`. One tip is that there may be no need to do "`u = v`" since `u` always points to the correct element.

Obviously, the correctness of `HeapPriorityQueue` depends on the correctness of `ArrayCompleteTree`. Please test your implementation of `ArrayCompleteTree` thoroughly before implementing `HeapPriorityQueue`. In the grading script, however, we will use the instructor's implementation of `ArrayCompleteTree` to evaluate your implementation of `HeapPriorityQueue`.

We provide you a skeleton of the `HeapPriorityQueue` class in `HeapPriorityQueue.h` with all private variables you need. You must use the skeleton to implement the priority queue. More importantly,

(1) You are not allowed to define any other member variables (public, private, or protected) in `HeapPriorityQueue`;
(2) You are not allowed to include any other header files except `iostream`, `stdexcept`, and `ArrayCompleteTree.h` that have already been included; and
(3) you are not allowed to define any public member functions or other classes in `HeapPriorityQueue.h`.

However, you are allowed to define any *private* member functions if you see fit. In fact, we recommend you to define the following two private member functions that you can use in several places in the code:

- `void bubbleUp(const Position& u)`
- `void bubbleDown(const Position &u)`

These functions implement the up-heap bubbling algorithm and the down-heap bubbling algorithm, respectively, in Section 8.3 of the textbook.

The `Position` object of `HeapPriorityQueue` is the same as the one for `ArrayCompleteTree`. Hence you can conveniently reuse the same `Position` class in `ArrayCompleteTree` by adding the following `typedef` statement to `HeapPriorityQueue`.

```
typedef typename ArrayCompleteTree<E>::Position Position
```

Notice that this statement is similar to the same statement in Code Fragment 8.14 on Page 349 of the textbook.

In `HeapPriorityQueue.h`, there is a class called `LessThanComparator`, which generalizes the LeftRight class on Page 326 in the textbook. `Simulator.h` and our grading script will use this comparator when defining a `HeapPriorityQueue` object. You do not need to modify this class.

Let's talk about the private member variables in `ArrayCompleteTree`:

- `ArrayCompleteTree<E> T`
- `C isLess`

Please see Code Fragment 8.14 on Page 349 of the textbook to see what they are. You are not allowed to define any other member variables in `HeapPriorityQueue`.

Let us talk about the set of all required public member functions that your class must support. You will have to implement the following public member functions in order to be accepted by our grading script.

- `HeapPriorityQueue<E,C>`'s public constructors:

  - `HeapPriorityQueue()`
  - `HeapPriorityQueue(const HeapPriorityQueue& q)`

- `HeapPriorityQueue<E,C>`'s public member functions:

  - `int size() const`
  - `bool empty() const`
  - `Position insert(const E& e)`
  - `const E& min()`
  - `void removeMin()`
  - `void remove(const Position& u)`
  - `Position replace(const Position& p, const E& e)`

As can be seen, these public member functions are the ones in Code Fragement 8.14 and Section 8.4 in the textbook. We also add a copy constructor since `Simulator.h` needs it. All the copy constructor does is to call the copy constructor of `ArrayCompleteTree`. There is no need to copy `isLess` since `isLess` has no member variable.

We need to talk about how to implement `remove()` because the algorithm is not presented in the textbook. An excellent description of the removal algorithm (i.e., the heap deletion algorithm) is presented in http://www.mathcs.emory.edu/~cheung/Courses/171/Syllabus/9-BinTree/heap-delete.html. In summary, the algorithm works follow:

- Step 1: Delete the entry referred by the `Position` object `u`.
- Step 2: Replace the deleted entry with the last entry in `ArrayCompleteTree`. Notice that the last entry is the "furthest right node" on the lowest level of the complete binary tree. Remove the last entry from `ArrayCompleteTree`. Let `u` be the entry after the replacement.
- Step 3 (the heapify step): if the key of the element of `u` is less than the key of its parent (if any), use the up-heap bubbling algorithm to push `u` upward in the tree; otherwise, use the down-heap bubbling algorithm to push `u` downward in the tree.

Please read the web page to see why we need to the heapify step. Moreover, please be aware of the situation in which `u` is the last entry; you do not need the heapify step in this situation.

Tips for debugging: As usual, you should debug your code thoroughly before submission. One common practice is to include some functions to check the integrity of your data structure. For `HeapPriorityQueue`, it would be helpful to check whether the heap property has been violated at any point. Clearly, these functions should be disabled or removed in the production code. Notice that the implementation of these functions is not required in this exercise.

Task 3: Implementing Hash Maps

One drawback of heap-based priority queues is the lack of an efficient support of element removal: While the heapify step takes only O(log n) running time, the time to locate an element in a heap is O(n). Priority queues based on balanced binary trees do not suffer from this problem because finding an element in a balanced binary tree is O(log n). However, balanced binary trees are often less space efficient than the heaps based on array implementation of complete binary trees. Therefore, binary heaps are still preferred if there is a way to avoid searching an element in a heap. In fact, in many applications, you know the `Position` object of the element you want to remove, as long as you store the `Position` object of the element somewhere when you insert the element into the priority queues. For this purpose, the textbook modifies `insert()` to return a `Position` object in adaptable priority queues. Then the next question how we can be keep track of the `Position` objects returned by `insert()`. We need a data structure that can associate an inserted element with the corresponding Position object. This data structure does not need to order the elements by any means. Obviously, the data structure that fits the bill is the map data structure as described in Section 9.1 of the textbook.

For node deletion in heap-based priority queues, it is sufficient to have a map data structure that supports O(log n) search, insertion and deletion operations. Such map data structure can be implemented using balanced binary trees. However, if we implement a map data structure using balanced binary trees, this defeats our purpose of avoiding balanced binary trees in priority queues at the first place. Luckily, there are map data structures that are much more efficient than the ones based on balanced binary trees. Hash maps are map data structures based on hash tables. A good implementation hash maps can offer O(1) *expected* running time in all operations.

In this exercise, you are asked to implement hash maps based on hashing with separate chaining as described in Section 9.2.7 of the textbook. Your implementation should faithfully follow the description in the textbook, except

(1) `Entry`, as defined in Code Fragment 9.1 on Page 369 of the textbook, should be an inner class of `HashMap` as defined in Code Fragment 9.6 on Page 387. Moreover, the private variable `_key` in `Entry` should be declared with `const` since our hash map always keeps the key constant. For the same reason, we remove the member function `setKey()` in `Entry`. Furthermore, the return type of `value()` should not be "`const V&`"; it should be "`V&`" (i.e., remove the `const` keyword) since the value of an entry can be updated via this reference.
(2) `operator[]` is supported for the convenience of users. It allows users to use "`h[key] = value`" to create an entry where `h` is a `HashMap` object and associate the value to the key. What `operator[]` does is to search for an entry with the given key. If the key is not found, it will create a new entry in the hash map with the given key and return the reference to the value of the entry; otherwise, it will retrieve the entry and return the reference to the value of the entry. Hence, it does not throw any exception even if the key is not found. Please read `Simulator.cpp` to see how we should use this operator.

Once again, we have already included a skeleton of the `HashMap` class in `HashMap.h.` However, you will have to define your own private variables and inner classes in `HashMap`. You must use the skeleton to implement the hash map.

(1) You are not allowed to define any other member variables (public, private, or protected) in `HeapPriorityQueue`;
(2) You are not allowed to include any other header files except `iostream`, `vector`, `list`, and `stdexcept` that have already been included; and
(3) you are not allowed to define any public member functions as listed below.

However, you are allowed to define any *private* member variables and *private* member functions if you see fit.

The following is the set of all required public member functions that your class must support in order to be accepted by our grading script.

- `HashMap<K,V,H>`'s public constructor:
  - `HashMap(int capacity = 100)`
- `HashMap<K,V,H>`'s public member functions:
  - `int size() const`
  - `bool empty() const`
  - `Iterator find(const K& k)`
  - `Iterator put(const K& k, const V& v)`
  - `void erase(const K& k)`
  - `void erase(const Iterator& p)`
  - `Iterator begin()`
  - `Iterator end()`
  - `V& operator[] (const K& k)`

`HashMap` has two inner classes: `Entry` and `Iterator`. The following is the set of all required public member functions that these inner classes must support in order to be accepted by our grading script.

- `HashMap<K,V,H>::Entry`'s public constructor:
  - `Entry(const K& k = K(), const V& v = V())`
- `HashMap<K,V,H>::Entry`'s public member functions:
  - `const K& key() const`
  - `V& value()`
  - `void setValue(const V& v)`
- `HashMap<K,V,H>::Iterator`'s public member functions:
  - `Entry& operator*() const`
  - `bool operator==(const Iterator& p) const`
  - `Iterator& operator++()`

In the textbook, `HashMap<K,V,H>::Iterator` has a public constructor. Strictly speaking, it should be a private constructor since no one except `HashMap` can create an `Iterator` object for itself. `HashMap` can have access to this private constructor because `HashMap` is declared as a friend of `HashMap<K,V,H>::Iterator`. Therefore, please declare the constructor of `HashMap<K,V,H>::Iterator` to be private.

In `HashMap.h`, we include a class called `IntHashComparator`, which is an integer hash comparator that defines a hash function as required in the Code Fragment 9.6 on Page 387 in the textbook. `Simulator.h` uses this hash function to convert a job ID into a hash value. As you can see, the hash function simply returns the integer argument without any modification. It is sufficient since the type of the key is an integer. For other data types, please refer to Section 9.2.3 in the textbook to see how to construct a hash function properly. In our grading script, we may test your code with a different key type and a different hash function.

Notice that hash maps based on hashing with separate chaining is not the most efficient implementation of hash maps. For students who want to learn more of hash table implementation, we encourage them to try a different approach to implement the map data structure, as long as the set of public member functions are the same.

Discrete-Event Simulation in `Simulator.h` and `Simulator.cpp`

A type of simulation called *discrete-event simulation* is an excellent example for showing what priority queues and hash maps can do. Simulation is a major application of computers. In industries such as car manufacturing, simulators have been used to validate a system and evaluate their performance and safety. A simulation of dynamical systems often proceeds by advancing one time step at a time. The simulation is similar to action video games in which the virtual worlds are usually updated 60 times per second (i.e., 60 fps). However, this type of step-by-step simulation is too slow for some applications, especially when the simulation is very long. Furthermore, the discretization of time lines can cause errors in simulation. A discrete-event simulation avoids these issues by simulating the occurrence of events that can cause of a change of state in the system. Then the simulation can jump from one event to the next one and assume the state of system remains unchanged between two consecutive events. This is a powerful assumption as the simulation can skip many time steps, making the simulation runs much faster. For more information about discrete-event simulation, please visit this Wikipedia page:
https://en.wikipedia.org/wiki/Discrete_event_simulation.

The core of a discrete-event simulation is an event queue, which is a priority queue that orders the upcoming events according to the ascending order of their time stamps (i.e., the time that an event will happen). The simulation's loop repeatedly removes the next event (i.e., the minimum element) from the priority queue until there is no more event in the queue. The next event can cause a change of the state of the system, which in turn may trigger a sequence of future events. For example, in our simulation, an "Add Print Job" event can trigger two more events: "Start Print Job" and "End Print Job". The former event designates the start of a print job and the time stamp should be the beginning of the job. The latter event designates the end of a print job and the time stamp should be the beginning of the job thus the duration of the job. Both events will be added to the event queue. A discrete-event simulation basically repeats the process of adding and removing events until there is no more event in the event queue. What we want from the simulation is to monitor the evolution of the system states in simulation to see whether the system works properly.

We provide you an implementation of a discrete-event simulation for simulating the assignment of jobs (which can be print jobs) to machines (which can be printers). The code can be found in `Simulator.h` and `Simulator.cpp`. The main function in `main.cpp` provides you a simple interface to interact with the simulator. In the following, we will give you an overview of the code.

The discrete-event simulation is implemented in the class `Simulator` in `Simulator.h`. The class has six private member variables:

- `current_time` – the current time of the simulation.
- `machine_num` – the number of machines.
- `busy_machine_num` – the number of machines to which jobs have been assigned.
- `event_queue` – the event queue.
- `job_queue` – the job queue.
- `job_status_record` – a table that records the current status of jobs.

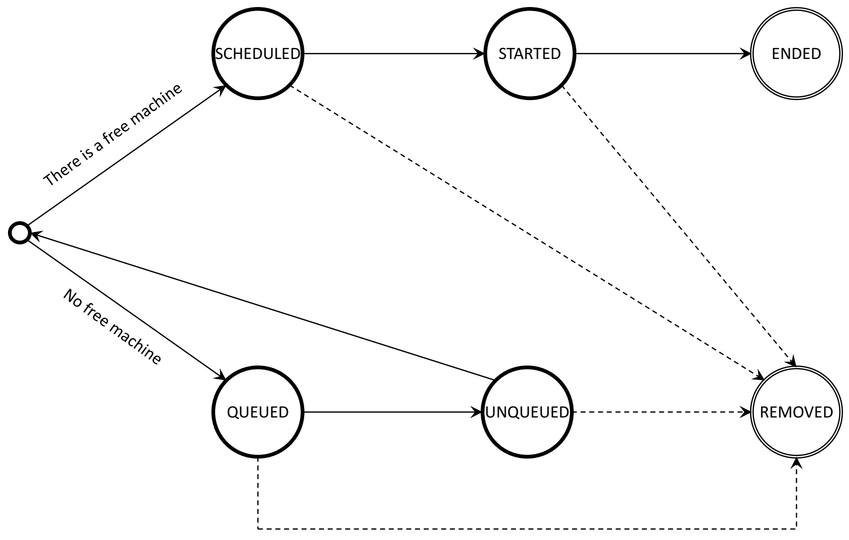
First off, a user must set the number of machines in the simulation via `setMachineNumber()`. After that, he can schedule some job requests via `requestAddJob()`, which takes three parameters: `request_time`, `job_id`, and `duration`. `request_time` is the time at which the request is made, and it must be no earlier than the current time of the simulation. job_id is an integer to identify the job, and it must be different from the job_id of other jobs that were added before `request_time`. `duration` is the time a machine must spend on the job if the job is assigned to the machine. Different jobs can have different `duration`. A call of `requestAddJob()` will insert an event `AddJobEvent` into the event queue, and the time stamp of `AddJobEvent` is `request_time`. `AddJobEvent` is defined in `Simulator.h`.

A user can also remove a job from the system via `requestRemoveJob()`, which takes two parameters: `request_time` and `job_id`. `request_time` is the time at which the request is made, and it must be no earlier than the current time of the simulation. job_id is an integer to identify the job. The result of calling `requestRemoveJob()` is that an event `RemoveJobEvent` will be created and inserted into the event queue. Notice that the `AddJobEvent` event of the job must occur before the `RemoveJobEvent` event of the job in the event queue; otherwise, the runtime error will occur later in the simulation.

The simulation starts by calling `run_next_event()`, which should be called repeatedly until `hashNextEvent()` is false. `run_next_event()` remove the minimum element from the event queue and send it to `eventHandler()`, which dispatches one of the four different event handlers according to the type of the event. Notice that there are four types of events in the event queues: `AddJobEvent`, `RemoveJobEvent`, `JobStartEvent`, and JobEndEvent. These events are defined Simulator.h. AddJobEvent and RemoveJobEvent are generated by requestAddJob() and requestRemoveJob(), respectively, as described before. JobStartEvent and JobEndEvent are generated by addJobEventHandler(), which is called by eventHandler() when the next event is JobStartEvent. addJobEventHandler() first checks whether there is any free machine. If so, the job will be directly assigned to the machine by assignJobToMachine(), which creates JobStartEvent and JobEndEvent and inserts them into the event queue; otherwise, the job will be added to the job queue (i.e., job_queue) by assignJobToJobQueue() and wait for a free machine later. The time stamp of JobStartEvent is the current time, while the time stamp of JobEndEvent is the current time plus the duration of the job. The jobs in the job queue are ordered according to the request time of AddJobEvent.

Apart from addJobEventHandler(), there are three other event handlers: removeJobEventHandler(), jobStartEventHandler(), and jobEndEventHandler().jobStartEventHandler() and jobEndEventHandler() are triggered by JobStartEvent and JobEndEvent, respectively, for bookkeeping. Both of them simply updated the status of the job in job_status_record. jobEndEventHandler() will also called a function freeMachine(), which releases a machine (by busy_machine_num--) and then immediately assign another job if the job queue is not empty.

Lastly, removeJobEventHandler()is called when the event is RemoveJobEvent. What it does depends on the status of the job. As defined in Simulator::JobStatus, there are six possible job status: QUEUED, UNQUEUED, SCHEDULED, STARTED, ENDED, and REMOVED. The following diagram shows how job status changes from one to another.



Upon receiving AddJobEvent, addJobEventHandler() will register the job by adding a record to job_status_record with a job status, which is `SCHEDULED` if the job has been assigned to a machine, or `QUEUED` if the job is put on the job queue. If a job is `SCHEDULED`, it will eventually become `STARTED` and then `ENDED`, unless the job is removed by `removeJobEventHandler()` upon receiving `RemoveJobEvent`, in which case the job status will become `REMOVED`. If the job is put on the job queue, the job status is `QUEUED`, and it will change to `UNQUEUED` if `freeMachine()` finds a free machine to which the job can be assigned. After that a new `AddJobEvent` is created at the current time and eventually the job status will turn into `SCHEDULED`. In any case, when `RemoveJobEvent` for the job is received, the events corresponding to the job will be removed from the event queue *or* the job will be removed from the job queue (if any). To facilitate the removal of the event and jobs from the priority queues, `JobStatus` maintains some position objects which refer to the locations of the event and jobs in `event_queue` and `job_queue`. These position objects are set when events are added to `event_queue` or jobs are added to `job_queue.`

For information about this discrete-event simulation, please read the code in `Simulator.h` and `Simulator.cpp`. In particular, please focus on how the code interacts with `HeapPriorityQueue.h` and `HashMap.h`, since you will have to implement `HeapPriorityQueue.h` and `HashMap.h` such that they work with the code of the discrete-event simulation.

Testing and Submission

All files that you need in this assignment are put in a zip file called `prog3.zip`, which contains the following files:

- `handout-3.pdf` — the handout of this assignment (i.e., this file).
- `ChangeLog.txt` — the change log.
- `desim-sample` — the sample program.
- `main.cpp` — the main function of your program.
- `ArrayCompleteTree.h` — your implementation of the array-based complete binary tree. You will submit this file.
- `HeapPriorityQueue.h` — your implementation of the heap-based priority queue. You will submit this file.
- `HashMap.h` — your implementation of the hash map. You will submit this file.
- `Simulator.h` — the declaration of the `Simulator` class and the events.
- `Simulator.cpp` — the implementation of the member functions in the `Simulator` class.
- `README.TXT` — your readme file. You will submit this file.

We provide you a sample program called “`desim-sample`” that implements all functionalities of this program. Please play with the program to see how it behaves. Since we have provided you the code of the simulator, the output of your program will be the same as the output of the sample program if your implementation of the priority queue and the hash map is correct.

We also provide two files that implement the discrete-event simulation: `Simulator.h` and `Simulator.cpp`. The code is well-documented and self-explanatory. Please take a look at the source code to see how it works.

In this exercise, you are ***not*** allowed to use any data structures in the Standard Template Library apart from the ones in the code skeletons in `ArrayCompleteTree.h`, `HeapPriorityQueue.h`, and `HashMap.h`. If in doubt, please contact the instructor to ask whether a data structure in STL can be used.

We will test your implementation of the complete tree, the priority queue, and the heap map *separately*. In other words, we will evaluate `ArrayCompleteTree.h` alone, without using your implementation of `HeapPriorityQueue.h` and `HashMap.h`. Likewise, we will evaluate `HashMap.h` alone. When we evaluate your `HeapPriorityQueue.h`, we will use the instructor's implementation of `ArrayCompleteTree.h`. Therefore, your implementation should adhere to the interface as specified in this handout.

In the behavior test, we will test all of your program together with `Simulator.h`, `Simulator.cpp`, and `main.cpp`. We will compile your programs to see whether they fit together. If the compilation is successful, we will check the behavior of the simulator.

We will test the completeness and correctness of your implementation, including whether your code will throw all necessary exceptions. We will also check whether your program will cause memory leak. The error messages in `runtime_error` exceptions do not have to be exactly the same as the messages in the sample program we provided. You can write the error messages in your own sentences. However, we will check whether your program throws the exceptions.

To compile your program on our submission servers, use this command:

```
g++ -o desim Simulator.cpp main.cpp
```

Before you submit your program, please check whether your program can be compiled correctly using this command on our submission server. We do not grade your program using other compilers.

Please also submit a plain text file called "`README.TXT`" to tell us the extent of your implementation, more specifically which parts have been implemented and which parts have not. Also, if there are some known bugs in your program, you should state them in the plain text file. An empty `README.TXT` is included in `prog3.zip`.

You will submit only four files: (1) **`ArrayCompleteTree.h`**, (2) **`HeapPriorityQueue.h`**, (3) **`HashMap.h`**, and (4) **`README.TXT`**. These files should be self-contained and you cannot submit additional files.

Please store these files in a directory called **assignment3** in the course project on Gitlab, which is set up according to our Gitlab guidelines. Please remember to push these files to Gitlab after you finish writing them. For more detail on how to use Gitlab, please read our Gitlab guidelines on our course website. Please email **Sangwoo Ha** at **swha@unist.ac.kr** if you fail to upload your files to Gitlab.

Automatic Grading and Late Submission Penalty

We developed shell scripts to grade your programs automatically without human intervention. In the past, many students didn't read this handout carefully and implemented their programs with wrong function names or filenames, etc., causing compilation errors. Our TAs spent too much time to help students to fix these issues in the past. In order to avoid wasting our TAs' time, we decide to give students opportunities to see the results of the grading script running with their programs *before* the deadline, such that students can fix any issues *themselves* before the final submission. We will start running the grading scripts at every midnight at least three days before the deadline. If you submit your program earlier, you will see the grading report generated by the grading script in the home directory in your account. Please fix any problems in your program according to the grading report, especially those that are caused by incorrect naming. *There will be a 10% penalty if our TA needs to help you to fix any issues related to the submission and automatic grading after the deadline.*

The scores in the pre-deadline grading reports are not final and they will not be recorded. However, the grading reports after the deadline are final. In fact, we will grade your programs three times after the deadline in order implement our late submission policy as described on our course webpage. The first real grading report will be generated right after the deadline. The second real grading report will be exactly one day after the deadline with a penalty of 15%. The third real grading report will be exactly third day after the deadline with a penalty of 30%. We will select the highest scores among these three real grading reports as your final score of this assignment. Clearly, if you do not update and resubmit your program after the deadline, the highest score will be the one right after the deadline. However, if you are not happy with the score on the first real grading report, you still have the chance to improve it by submitting your programs again. Notice that late submission will never decrease your final score.

Our grading script is evolving and hence we reserve rights to regrade your programs with a different script in a later time. Hence, the scores you see in the grading reports may not be final. Please report any bugs in our grading script if you find them.

Bug Reports

Please report any bugs in the codes we provide as well as any typos and errors in this handout and the grading reports to **Prof. Tsz-Chiu Au** at **chiu@unist.ac.kr**. We will look into the bug reports and fix the problems. For any other problems, please do not contact the instructor directly; instead, please email our TA who should be able to help you. Please cc the email to the instructor so that he also knows what is going on between you and the TA. Notice all emails between you, the TA, and the instructor should be written in English.

If we have to release a new version of the codes to fix the bugs, we will announce it on our course webpage. Before submitting your program, you should check the announcement to see whether the codes have been updated. Please make sure that your program works with the latest version of the codes we provide.