CPS 221: Computer Science II
Process Scheduling

Important Dates
Checkpoint: 5:00pm, Wednesday, 10/26
Final deadline: 5:00pm, Wednesday, 11/30

Points be evaluated based on working code, good coding practices, comments, etc.

Checkpoint: 50pts

- Test Files 10pts
- ArrayList 15pts
- LinkedList 15pts
- Process 5pt
- Simulate/schedule simulate 5pt

Final: 100pt

- Test Files 20pts
- ArrayList 10pts
- LinkedList 10pts
- BSTMultimap 30pts
- Process 5pt
- Schduler 5pts
- Simulate 5pt
- Schedule Simulation 5pt
- Timings / pdf 10pts

Total: 150pts

Introduction

You are so used to running multiple programs at once on your computer that you probably take that ability for granted. But most computers that you interact with only have 1-4 processors in them. How can they be running hundreds of programs simultaneously? In reality, most processes are *not* running simultaneously; the operating system actually switches between them very quickly, each process gaining exclusive control over the CPU for small amounts of time. The part of the operating system that decides what process should get to use the CPU (and for how long) is called the *scheduler*. In this project you will implement two common scheduling strategies using various data structures and compare their strengths and weaknesses.

Part 1: Round Robin Scheduling

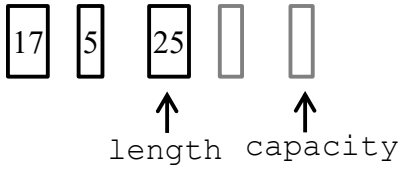
In this checkpoint you will implement one of the simplest scheduling strategies, *round robin scheduling*. This algorithm keeps processes on a queue and cycles through them, granting each process

a (small) fixed amount of time on the CPU before pausing it and moving on to the next process.

1) Generic Array List
Reading:
Chapter 2, Chapter 3

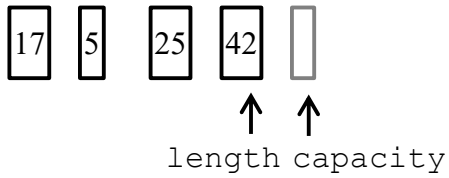
One simple way to implement a queue is to use a list! Python lists and C++ vectors are much more flexible than arrays, particularly because they can be dynamically resized. That said, the Python interpreter is written in C, so those lists must be built upon the basic constructs we have been studying! An *array-backed list* (also called a *dynamic array* or an *array list*) allocates a large array and stores the contents of the list inside it, keeping track of both the length of the list (`length`) and the size of the array (`capacity`).

For instance, here is a 3 element list stored inside a 5 element array:



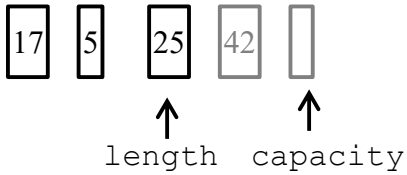
If the user wants to append 42 to the end of the list, this can be accomplished efficiently by incrementing `length` and placing the 42 in the appropriate location.

```
list.pushBack(42)
```



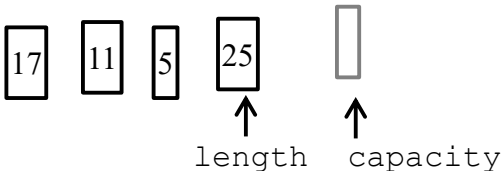
To pop the last element off of the list, simply decrement `length`.

```
list.popBack()
```



Inserting or removing elements from the middle of the list is less efficient. It involves shifting all of the elements after the location in question (either to make room or to fill in the gap).

```
list.insert(11, 1)
```



All this is well and good, until `length` exceeds `capacity`. At that point, we need a new array! Common practice is to create a new array that is double the size of the old one, so there is plenty of room for the list to grow. Then the contents of the list are copied over to the new array.

```
list.pushBack(98)
list.pushBack(56)
```

The class template `ArrayList` is declared in the provided file *ArrayList.hpp*. In *ArrayList.tpp* implement the class template (remember templates must be declared and defined in the same file).

As you create your class, keep the following guidelines in mind:

- Most of the methods take and return const references to items, rather than items themselves. This is to prevent unnecessary copying in the case that the items are large objects.
- Remember that a template is not code; it is a recipe to write code (with specific types filled in). So a template is never really compiled until a program uses it. Just trying to compile *ArrayList.h* will accomplish very little.
- So...**write a test program (*ArrayList_TEST.cpp*) and compile and test early and often!** Waiting until you've written the entire class before you try to compile and test is asking for a huge pile of compiler errors and bugs to sort through. Please implement your tests using only the provided public interface to the class (do not add or alter public methods).
- Make sure you test your `ArrayList` with more than one type to ensure that your template is sufficiently generic (maybe try `int`s and `string`s).
- Focus on efficiency: make each method as computationally efficient as you can. None of these should take more than linear time, and some can be done in constant time.
- Make sure you do not create any memory leaks. When you are done with something you dynamically allocated, delete it.
- Avoid code duplication: though you may not add public methods, you *should* consider adding private helper methods that capture common functionality in the public methods. You may also consider implementing some methods by calling other methods.
- You are not responsible for error checking. If a user accesses a location that is not in the list, you are not required to handle it gracefully. Typical C++ convention is to do as you are told and not ask questions (i.e. sacrifice safety for the sake of efficiency and programmer choice).

2) Inheritance

The provided file *List.hpp* contains an abstract base class template `List`. Modify your `ArrayList` class in the following ways.

- Make it a subclass of `List`.
- Make all of its methods `virtual` (except the constructors).
- Change its `private` members and methods to `protected`.

Note: all of these changes are made in the declaration of the class!

3) Processes

You will now implement the round robin scheduling algorithm. You've been provided with the files *Process.hpp/cpp*, that define three classes to represent processes in the scheduler. The `Process` class is the abstract base class. The two concrete classes represent different types of process:

- `CPUBoundProcess` represents a process that just has a lot of work for the CPU to do. This type of process will always use any CPU time it is given.
- `IOBoundProcess` represents a process that spends a lot of time doing input/output (I/O). For instance, it might be interacting with a user, or reading/writing files to the disk. As a result, it spends a lot of time waiting for I/O and can't always use the CPU, even if it is available. When a

process is stuck waiting for I/O, we say it is *blocked*.

Review these classes and make sure you understand them. To demonstrate your understanding, add a comment above each function in *Process.cpp* that describes what the function does: what does it take, what effects does it have, and what does it return?

4) Round Robin Scheduler

In *Scheduler.hpp* declare an abstract base class (not a template!) called `Scheduler`. It should have the following pure virtual public methods.

- `void addProcess(Process* proc)` – adds the given process to the processes to be scheduled
- `Process* popNext(int curCycle)` – removes and returns the next process in the schedule. (Returns 0 if there is no available process).

`Scheduler` should also have a virtual destructor that does nothing. Also implement a concrete subclass of `Scheduler` called `RoundRobin`. It should have one protected member variable called

- `List<Process*>* procQueue`

and the following public methods (make them all virtual, except the constructor).

- `RoundRobin()` – initializes `procQueue` as an empty `ArrayList`.
- `~RoundRobin()` – deletes `procQueue` (not its contents!)
- `void addProcess(Process* proc)` – adds the given process to the back of `procQueue`.
- `Process* popNext(int curCycle)` – returns the first un-blocked process in the queue (moving any blocked processes to the back of the queue) or 0 if all processes are blocked.

Implement this class in *Scheduler.cpp.* You should implement *RoundRobin_TEST.cpp* to test your class to be sure it does what you think it does. You'll notice that `Process` objects have an ID. It has no functional purpose, but you can use it in testing to see if processes come out of the scheduler in the order you expect. Note that, though `IOBoundProcess` objects behave randomly, you can obtain a blocked process to test with by repeatedly called `useCPU` until it becomes blocked.

5) Scheduling Simulation

You will now implement a simulation of process scheduling. Roughly speaking, the basic unit of computation time is the *CPU cycle*. One cycle consists of the processor fetching an instruction, and then carrying it out. You've probably seen a processor's performance measured in gigahertz (GHz). This number is meant to represent how many billions of cycles are performed every second. So a 3.5GHz processor performs 3.5 billion cycles every second. Now you know!

Your simulation will simulate the CPU for a certain number of cycles. It will repeatedly ask the scheduler for a process, then allow that process access to the simulated CPU for a small number of cycles (called a *slice*). Sometimes the process will use the whole slice. Sometimes it will use only part of it (or none of it!) due to I/O blocking. Along the way you will keep track of some statistics to evaluate the performance of your scheduling algorithm. `Process` objects keep track of the number of simulated CPU cycles that the process has used (CPU time) and the number of simulated cycles in which the process was ready to use the CPU but had to wait (wait time).

One thing we will want to measure is the actual runtime of the simulation (to measure how efficient the scheduler is). The C++11 standard introduced some new time measurement tools that are fairly complicated. Luckily we only need the basics. To use these features, you will have to give g++ the -

std=c++11 option when you compile your code.

If you `#include <chrono>` you will gain access to many time-related functions. To get the current time from the system clock, you can write:

```
auto t = chrono::system_clock::now();
```

The `auto` keyword is a handy addition in C++11. It means "the type that this function returns." This can save you some typing (in more ways than one!). Abstractly, `t` in this scenario is a "time point". If you subtract time points, you get a "duration":

```
auto dur = t1 - t2;
```

If you want the duration to be measured using particular units, use a new type of casting operator:

```
auto durMS = chrono::duration_cast<chrono::milliseconds>(dur);
```

To get the number of ticks of the appropriate unit contained within a duration, call the count method:

```
int elapsed = durMS.count();
```

Now you can write your simulation. In *simulate.cpp/hpp*, implement the function:

```
double* simulate(Scheduler* sched, int numCPUBound, int numIOBound, int numCycles).
```

The goal of this function is to simulate the CPU scheduler with the specified number of CPU-bound and I/O-bound processes. In the end it will return an array containing some important statistics. First, it will report the number of real-world nanoseconds the simulation took divided by the number of times `popNext` was called. This measures the efficiency of scheduler operations. Next it will report the average total CPU time and average total wait time of CPU-bound and I/O-bound processes after the simulation. This will show how fairly the scheduler distributes CPU resources. Some notes:

- The simulation should allocate processes and add them to the scheduler.
- The main loop of the simulation should go until the number of simulated cycles passes `numCycles`. In each iteration...
  - Get the next process from the schedule.
  - If the scheduler returns a null pointer, just increment the cycle number by 10. Otherwise...
  - Call the process' `useCPU` method with the current cycle number and slice size of 10.
  - The `useCPU` method returns the number of cycles used by the process (which may not be the entire slice). Add that to the number of simulated cycles so far.
  - Add the process back into the scheduler.
- When the simulation is over,
  - Allocate and fill a 5 element array that contains (in this order):
    - The number of actual nanoseconds the simulation took divided by the number of times you used the scheduler. This gives a sense of how much overhead the scheduler causes on each insertion/removal of a process from the data structure.
    - The average CPU time of all CPU-bound processes (using the `getCPUTime` method)
    - The average wait time of all CPU-bound processes (using the `getWaitTime` method)
    - The average CPU time of all I/O-bound processes.
    - The average wait time of all I/O-bound processes.
  - Prevent memory leaks by cleaning up anything you dynamically allocated in this function.
  - Return your array of statistics.
- In order to gather these statistics you will need to keep track of which processes are of which type. There are many ways to accomplish this – it's up to you!
  - Note, however, that you can't rely on the scheduler using the round robin strategy – you will soon be implementing other schedulers. Specifically, you should not count on `popNext` to let you go through all the processes in order, since other schedulers might not support that!

In *schedulesim.cpp* implement a `main` function, which should take three command-line parameters: the number of CPU-bound processes, the number of I/O-bound processes, and the number of cycles to simulate. It should create a `RoundRobin` scheduler and pass it (and the other parameters) to the `simulate` function. Finally it should nicely print the results of the simulation with clear labels. Try out your simulation with different parameters. You don't need to unit test the `simulate` function, but you can do some sanity checks using *schedulesim*.

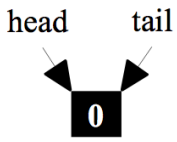
- For CPU-bound processes, you would expect average CPU time + average wait time to be roughly equal to the total number of cycles (which may be slightly higher than `numCycles`).
- For I/O-bound processes, you would expect average CPU time + average wait time to be less than the total number of cycles (since they spend some time being blocked).
- If you have all CPU-bound processes, the average CPU time should be roughly the total number of cycles divided by the number of processes.

As you try the program out, you will notice that the time per scheduler call increases with the number of processes. Round robin scheduling lies at the core of the *O(1) Scheduler*, which was used in the Linux kernel for many years. Based on the name, if our scheduler is O(*n*) we're doing something wrong! As you know, array-backed lists are not the best choice for implementing a queue.

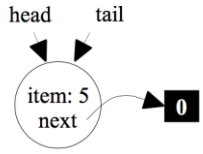
6) Linked List
Reading: Chapter 4, Chapter 5

In order to implement an efficient queue, you'll need a linked list! So, you will implement a `LinkedList` class template that also inherits from `List`.
There are a number of ways to implement a linked list, particularly when it comes to handling the ends of the list. In your implementation, each node in the list will maintain a pointer, `next`, which points to the next node in the list. The list itself will maintain two pointers, `head` (to the first node in the list) and `tail` (to the last node in the list). Use a null pointer to indicate the end of the list: the `tail` node should have 0 as its `next`. When the list is empty, the list should have no nodes; both `head` and `tail` should be 0.

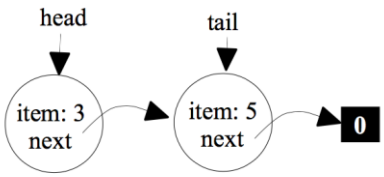
To illustrate, here is a picture of an empty list:



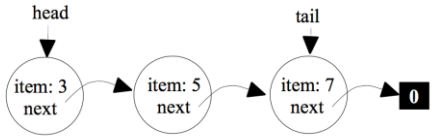
Now say 5 is inserted to the list (whether from the front or back):



Now say 3 is pushed onto the front of the list:



And finally say 7 is pushed onto the back of the list:



To save you some typing time, you've been given *LinkedListNode.hpp/tpp*, which defines the class template `LinkedListNode`. It stores an `item` and a `next` pointer (with accessor/mutator methods).

In *LinkedList.hpp/tpp*, implement the class template `LinkedList`. It should inherit from `List`, and therefore it should implement all of the pure virtual methods (please make your methods virtual too!). It should also have:

- Three protected class members
  - `head` – a pointer to the first node in the list
  - `tail` – a pointer to the last node in the list
  - `size` – an `int` keeping track of how many items are in the list
- A default constructor that creates an empty list.
- A destructor that deletes *all* nodes in the list.

As you did when you developed your array-backed list, please keep the following guidelines in mind:

- **Test every method in *LinkedList_TEST.cpp* you write when you write it.**
  - Write stubs for methods you have not implemented so you can compile and test your code.
  - **Waiting until you have written your entire class template to compile and test it is a terrible idea and will cause unnecessary pain.**
  - Again, please use the given public interface for testing. Do not alter or add public methods.
- Avoid code duplication by creating protected helper methods to perform common tasks.
- Be careful about the head and tail pointers! You may have to treat lists with 0 or 1 items as special cases in some functions.
- Watch out for memory leaks. When you remove a node from the list, make sure you delete it.
- Again, you do not have to do any error checking.

7) Return to the Round Robin Scheduler

Now you'll want to implement round robin scheduling using your linked list instead of your array-backed list. One idea would be to copy/paste your code into a new class. Hopefully that feels

ridiculous. A better idea would be to use inheritance to avoid code duplication.

In *Scheduler.hpp/cpp* create a class `FastRoundRobin` that is a subclass of `RoundRobin`. It should inherit everything (so don't override any methods)! It just needs to define is its constructor, which should delete the list that `RoundRobin` created (remember that the superclass' constructor is always invoked first) and point `procQueue` to an empty `LinkedList`. Alter *RoundRobin_TEST.cpp* to test the new scheduler as well as the old one. It should behave exactly the same, but faster.

Now alter the `main` function in *schedulesim.cpp* so it runs the simulation with both the `RoundRobin` scheduler and the `FastRoundRobin` scheduler. You should avoid code duplication. More specifically...

- Create an array of 2 `Scheduler*`s. It should contain a pointer to a `RoundRobin` scheduler and a pointer to a `FastRoundRobin` scheduler.
- You'll also want to clearly label your output. Create an array of 2 `string`s. These will be used to label the simulation results using each scheduler so make them brief but descriptive.
- Use a loop to set up and run the simulation with each scheduler and clearly present the results for each one.
    - Make sure you don't leak any memory!

Now that the time complexity issue is dealt with, you'll notice another issue with the round robin scheduling algorithm: I/O-bound processes receive less CPU time than CPU-bound processes. This is because they rarely use the entire time slice given to them due to I/O blocking. The I/O-bound processes also have long wait times – when the process becomes unblocked, it still has to wait until it comes back around in the queue to use the CPU again, which could cause frustrating pauses and stutters for users. Ideally, I/O-bound processes could get access to the CPU soon after they become unblocked, so they can quickly get to their next I/O request.

The O(1)-Scheduler dealt with the above problem with multiple queues and complicated schemes that (attempted to) determine based on behavior whether processes were CPU-bound or I/O-bound, and then adjust the order. For the next checkpoint you will (start to) implement a more elegant solution.

8) Data

Run your simulation with increasing numbers of CPU-bound processes to observe its behavior. Specifically, your simulation should simulate 1,000,000 CPU cycles and have 10 I/O-bound process. Run it with 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, and 10,000 CPU-bound processes. Record the time per scheduler call results for each simulation and plot them together on the same graph (they should look very different). Also, for each run record the ratio of the average CPU time for I/O-bound processes to the average CPU time for the CPU-bound processes. If the scheduler fairly distributes CPU cycles this ratio should close to 1. Plot these together on another graph (they should be very similar for both schedulers). Submit your graphs in PDF files *timing.pdf* and *cputimes.pdf*. Make sure your graphs are readable, nicely formatted, and clearly labeled.

---You have reached Checkpoint ----

Files to turn in by 5:00pm, Wednesday 10/26:

List.hpp, ArrayList.hpp/tpp, LinkedList.hpp/tpp, LinkedListNode.hpp/tpp Process.hpp/cpp, Scheduler.hpp/cpp, simulate.hpp/cpp, schedulesim.cpp, ArrayList_TEST.cpp, RoundRobin_TEST.cpp, LinkedList_TEST.cpp, catch.hpp, Makefile, timing.pdf, cputimes.pdf

Part 2: Completely Fair Scheduling

In the last part you implemented basic round robin scheduling using a linked list. As you saw, the round robin scheduler has some problems – in particular I/O bound processes tend to get fairly little CPU time and spend a lot of time waiting, which can cause frustration for users.

These days, Linux uses the *Completely Fair Scheduler*, which handles this problem in an elegant manner. The idea is to maintain a container of processes sorted by how much CPU time they have been given so far and to always pick the process with the least CPU time. This way, if an I/O-bound process uses the CPU for a short time and then blocks, when it unblocks it will still be close to the front and can use the CPU when it needs it. The main data structure used to accomplish this is a binary search tree. To implement completely fair scheduling, we will exploit the fact that if you do an in-order traversal of a BST, you can read off the elements in sorted order. However, note that the key for each process will be the CPU time used by the process, and that might be the same for multiple processes! Thus, we will need to build a *multimap* that allows many values to associate to the same key.

9) Binary Search Tree Node

In *BSTNode.hpp/tpp* implement the class template `BSTNode`. It should take two template parameters, `key_t` and `val_t`, which represent the types of keys and values, respectively. A node should store its key and value, pointers to its children, and a pointer to its parent. It should have these public methods:

- A constructor that takes a key and a value (as const references!), copies them into its members and initializes the children and parent pointers to 0.
- `get/setValue` – return/take a const reference to the node's value (of type `val_t`)
- `get/setKey` – return/take a const reference to the node's key (of type `key_t`)
- `get/setLeftChild` – return/take a pointer to the left child
- `get/setRightChild` – return/take a pointer to the right child
- `get/setParent` – return/take a pointer to the parent

Test your node class in *BSTNode_TEST.cpp.*

10) Creating a Binary Search Tree

In *BSTMultimap.hpp/tpp* implement the class template `BSTMultimap`, which takes two template parameters: `key_t` and `val_t`. The class should have three members:

- `BSTNode<key_t, val_t>* root` – the root of the tree
- `BSTNode<key_t, val_t>* sentinel` – a special value that means “no node.” That is, a node that does not have one or both children should point to the sentinel; the root's parent should be the sentinel; and an empty tree should have the sentinel as its root. For now this will be set to 0, though by making it a variable we have the freedom to change it later on.
- `int numItems` – the number of key/value pairs in the tree

Add the following public methods (you may add protected helpers, if you wish). All but the constructor should be virtual. Note that in all cases, you should use the `sentinel` variable to refer to a non-node,

and not the value 0.

- The constructor takes no parameters and initializes `sentinel` to 0, `root` to `sentinel`, and `numItems` to 0.
- `void insert(const key_t& key, const val_t& val)` – Creates a new node with the given key/value pair and puts it in an appropriate location (see Cormen's TREE-INSERT algorithm).
- `int getSize()` – Returns the number of items in the tree.
- `bool isEmpty()` – Returns `true` if the tree is empty, `false` otherwise.
- `string toString()` – For testing/debugging purposes (will not be graded). This should represent the nodes in your tree in such a way that you can tell whether they have been placed correctly! I recommend adding them to the string in a breadth-first manner (though you may have other ideas). To perform a breadth-first traversal:
  - Make a queue (use the STL `queue` template) with the root in it. Until the queue is empty...
  - Add the key and value of the node at the front of the queue to the string (use to_string!).
  - Enqueue the node's children.

Remember, *do not imagine that you can literally translate pseudocode into C++ and be done.* Fully understanding what you are writing will help you avoid errors and to catch them when they occur! You should also adhere to code style guidelines, even when the pseudocode does not. In particular, make sure you use informative variable names and comments to clarify the flow of your code.

Stop and implement *BSTMultimap_TEST.cpp* to thoroughly test your class template so far. Build some trees by hand and make sure your class template builds the same trees.

11) Search and Destroy

Now add the following virtual public methods (again, feel free to use protected helpers if you wish):

- `bool contains(const key_t& key) const` – Returns `true` if the given key is contained in the tree and `false` otherwise (see Cormen's TREE-SEARCH algorithm).
- `void clear()` – Empties the tree: deletes all nodes and resets `root` and `numItems`. *Hint: you had to go through all the nodes to print the tree too.*
- The destructor should simply call `clear()`.

Again, test your new functions thoroughly!

12) Binary Search Tree Iterator

We will need a way to go through the elements in the BST in order. To do that, we will use an iterator: an object that provides an interface for going through the elements of a data structure one-by-one. You've been provided with *BSTForwardIterator.hpp* which declares the class template `BSTForwardIterator`. You'll have to implement it in *BSTForwardIterator.tpp*. Some notes:

- The iterator has two members
  - `current` – the node the iterator is currently pointing to
  - `sentinel` – the pointer value that represents the end of the tree (this will be 0 for now, but we are making this a variable for the purpose of flexibility in the future)
- The `next` method should set current to the next node in an in-order traversal of the tree.

    - You may wish to refer to the TREE-SUCCESSOR algorithm in Cormen.
    - When, for instance, checking if a node has a child, you should compare its pointer to `sentinel` (and not explicitly compare to 0).
    - If there is no successor (the iterator is pointing to the last node in the traversal), the method should set `current` to `sentinel`.
    - If `current` is `sentinel`, the method should do nothing.
- The `isPastEnd` method should return `true` if the current node is the `sentinel` value (and therefore the end of the data structure has been reached by the iterator).
- The iterator allows one to `get` the key and value of the node it points to, but only `set` the value (changing the key could mess up the tree!).
- The last line in the class declaration defines `BSTMultimap` as a *friend class* to the iterator. In C++, a friend can access a class' private members. In particular, this will allow the BST data structure to see what node an iterator is pointing to, but not users of the iterator.

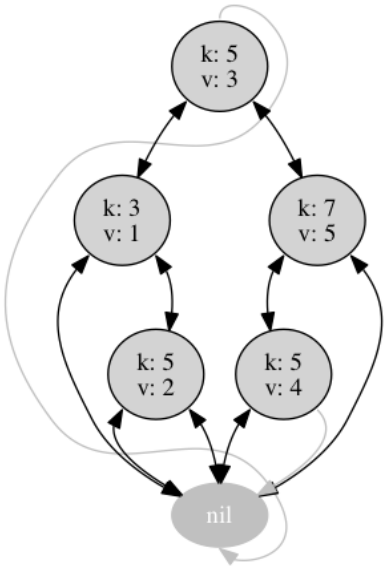
Now add the following virtual public method to `BSTMultimap`:

- `BSTForwardIterator<key_t, val_t> getMin() const` – Returns an iterator pointing to the node with the minimum key in the tree, which is also the first node in an in-order traversal of the tree (see Cormen's TREE-MINIMUM algorithm). Notes:
    - If the tree is empty, return an iterator pointing to the sentinel.
    - Iterators are a public interface; you don't generally need to use them internally because you have direct access to the nodes themselves. Find the minimum node, and only then package it up in an iterator to return to the user.
    - Note that this method returns an iterator object, not a pointer to a dynamically allocated iterator object. Because an iterator is very small (it just contains two pointers), it's no big deal to copy one. Passing/returning iterators by value is inexpensive and frees the user from having to remember to delete dynamically allocated iterator objects.

You should now be able to thoroughly test your iterator (within *BSTMultimap_TEST.hpp*). Create example trees and make sure it really does produce an in-order traversal. Make sure it behaves well near the ends of the tree too!

13) More Searching

Add the following virtual public methods to `BSTMultimap`:



- `BSTForwardIterator<key_t, val_t> getMax() const` – Returns an iterator pointing to the node with the maximum key in the tree, which is also the last node in an in-order traversal of the tree (the reverse of `getMin`). If the tree is empty, return an iterator pointing to the sentinel.
- `BSTForwardIterator<key_t, val_t> findFirst(const key_t& key) const` – Returns an iterator pointing to the *first* node (in an in-order traversal) that has the given key. If the key is not present, return an iterator to the sentinel.
- `BSTForwardIterator<key_t, val_t> findLast(const key_t& key) const` – Returns an iterator pointing to the *last* node (in an in-order traversal) that has the given key. If the key is not present, return an iterator to the sentinel.

These operations should be done in O(*h*) time, where *h* is the height of the tree. You'll want to spend some time with some examples to develop a procedure for `findFirst` and `findLast`. Since they can be done in O(*h*) time, you can bet that they involve some variation on the usual BST search procedure (do not simply use an iterator to search the tree linearly!). Note that one of the two could be simpler than the other, depending on how your `insert` works! Still, you should write them both in a way that works for any legitimate binary tree, regardless of the specifics of the `insert` function. One way to deal with this is a Red-Black or AVL Tree that are talked about in Algorithms (Chapter 10 in this book), but we will not go into this on this project.

14) More Destroying

Finally, add the following virtual public method:

- `BSTForwardIterator<key_t, val_t> remove(const BSTForwardIterator<key_t, val_t>& pos)` – Removes the node pointed to by the given iterator and returns an iterator pointing to that node's successor in an in-order traversal (which might be the `sentinel`).
    - Be sure you find the successor *before* you remove the node from the tree!
    - Iterators provide a handy way to get a successor node. Other than that, you should probably deal directly with nodes as much as possible. Because `BSTMultimap` is a friend class to `BSTForwardIterator`, you can directly access the iterator's `current` member.

I highly recommend that you work out exactly what you want to do on paper before you implement this. And, as always, make sure you test what you wrote for correctness!

15) Completely Fair Scheduler

In *Scheduler.hpp/cpp* implement a concrete subclass of `Scheduler` called `CompletelyFair`. It should have one protected member variable called

- `BSTMultimap<int, Process*>* procTree`

and the following public methods (make them all virtual, except the constructor obviously).

- `CompletelyFair()` – initializes `procTree` as an empty `BSTMultimap`.
- `~CompletelyFair()` – deletes `procTree` (not its contents!)

- `void addProcess(Process* proc)` – adds the given process to the tree with the process' current CPU time as its key.
- `Process* popNext(int curCycle)` – First obtain an iterator pointing to the process with the minimum CPU time. Then use the iterator to traverse the tree until you find a process that is not blocked. Remove and return that process.

Test your class in *CompletelyFair_TEST.cpp.* Once you are confident that it works, you should be able to add your completely fair scheduler to the loop in *schedulesim.cpp*. When you test it out you should see that I/O bound processes are treated far better. They should have lower wait times and higher CPU times overall. Time complexity...is another story.

16) Data

Again run *schedulesim* for 1,000,000 cycles with 10 I/O-bound process and 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, and 10,000 CPU-bound processes. Plot the time per scheduler call results from all three schedulers on one (well-labeled, nicely formatted) graph. Also plot the ratio of average I/O-bound CPU time to average CPU-bound CPU time with all three schedulers. Submit your graphs in PDF files *timing.pdf* and *cputimes.pdf*.

As you'll see in your graphs, completely fair scheduling lives up to its name; it is much more fair in distributing CPU cycles. However, the BST is extremely inefficient. Tree operations seem to take essentially linear time, which means the tree must be extremely imbalanced. Can you figure out why it becomes so imbalanced? In the next part you will greatly improve the efficiency of the completely fair scheduler using a self-balancing tree structure.

---Final Turn In---
Files to turn in by 5:pm, Friday 11/30:
Turn in all files!