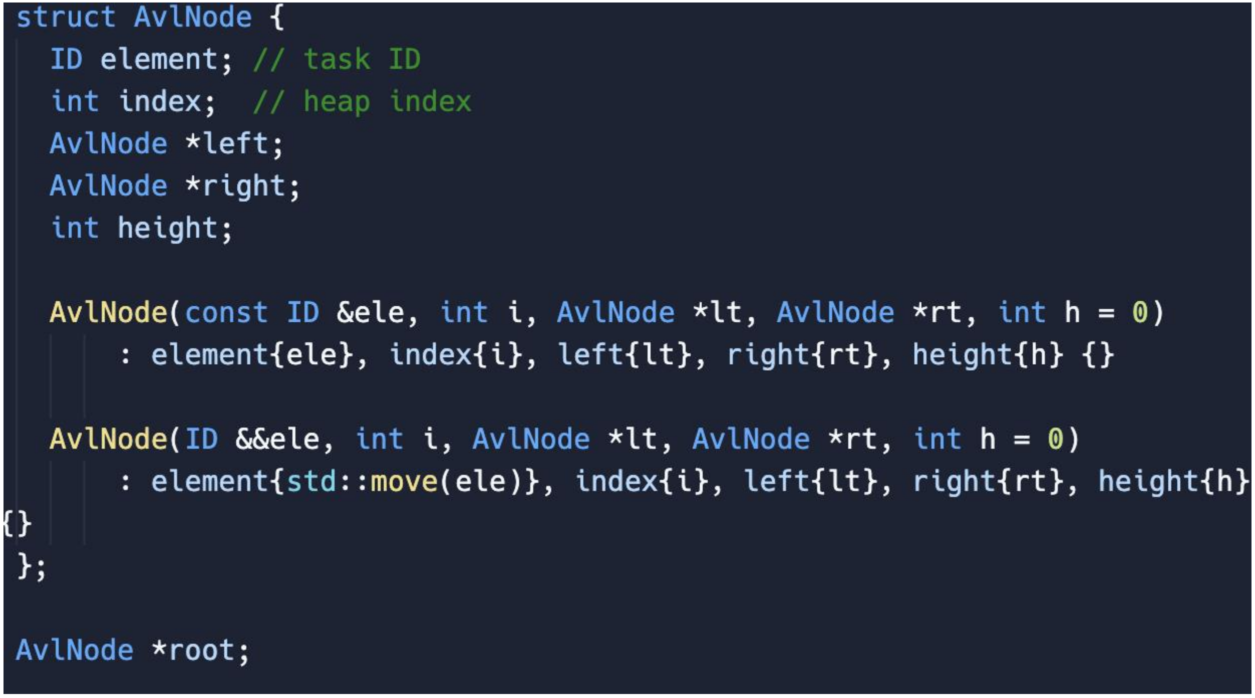
CMPT 225 D100 Assignment 3 Reports

Mirrien Liang (301325351)

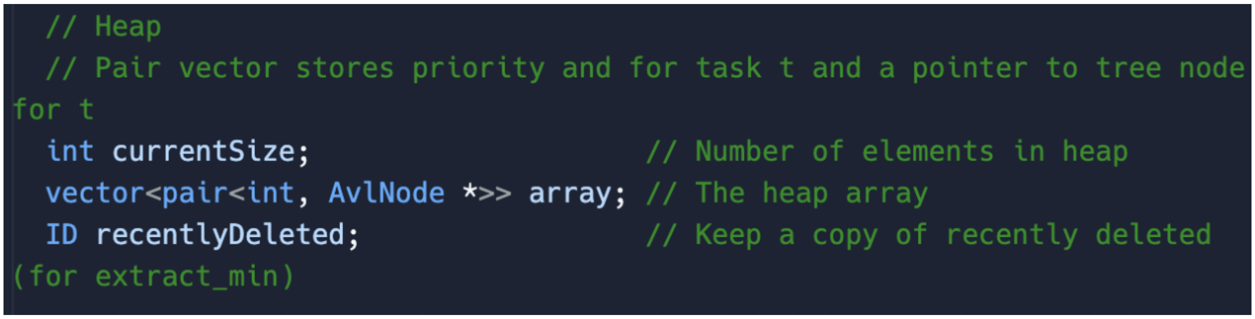
The Priority Queue has only one class *PQ*. It has the following data members:

1. AVL Tree
    a. AVL nodes used to construct embedded AVL tree. A definition of an AVL node is:
        i. ID-typed Task ID is stored in *element*
        ii. Each AVL node also contains an integer *index*, corresponding to the position of that task in the min-heap array
        iii. As usual AVL nodes would contain, there are **left*, **right*, and *height* contained in each node for tree construction
    b. A root pointer for AVL tree



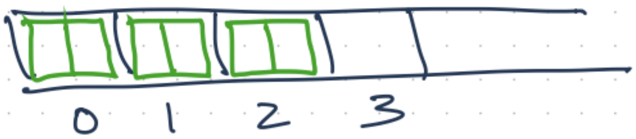
```
struct AvlNode {
  ID element; // task ID
  int index;  // heap index
  AvlNode *left;
  AvlNode *right;
  int height;

  AvlNode(const ID &ele, int i, AvlNode *lt, AvlNode *rt, int h = 0)
      : element{ele}, index{i}, left{lt}, right{rt}, height{h} {}

  AvlNode(ID &&ele, int i, AvlNode *lt, AvlNode *rt, int h = 0)
      : element{std::move(ele)}, index{i}, left{lt}, right{rt}, height{h}
{}
};

AvlNode *root;
```

2. Binary Min-Heap (Array-Implemented)
    a. A size counter *currentSize* tracking the number of pairs in the array
    b. An array (vector) of pairs where each pair contains:
        i. a priority integer for a task *t*
        ii. a pointer pointing to an AVL tree node that corresponds to the task linked with that priority
    c. A temporary ID-typed container *recentlyDeleted* storing a copy of recently deleted task ID. This is used as a convenient container to store returned values in function *deleteMin()*. May be of some other creative usages…

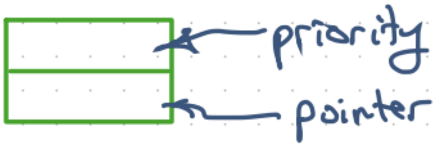


```
  // Heap
  // Pair vector stores priority and for task t and a pointer to tree node
for t
  int currentSize;                      // Number of elements in heap
  vector<pair<int, AvlNode *>> array; // The heap array
  ID recentlyDeleted;                   // Keep a copy of recently deleted
(for extract_min)
```

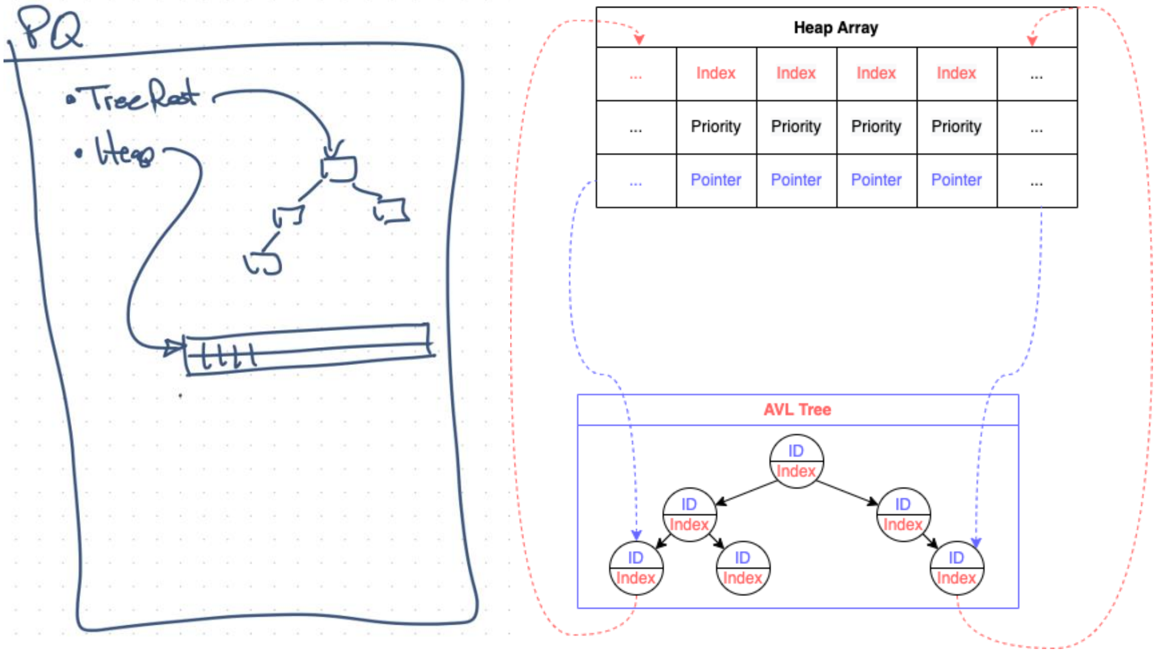
③ array of pairs.



pair =



A visualization of the PQ class with embedded AVL tree and Heap Array:


PQ
• TreeRoot
• Heap
Heap Array
... Index Index Index Index ...
... Priority Priority Priority Priority ...
... Pointer Pointer Pointer Pointer ...
AVL Tree
ID Index
ID Index
ID Index
ID Index
ID Index
ID Index

The PQ class has the following accessing methods:

1. All AVL tree nodes must be accessible by using the pointers stored in Heap Array.
2. Heap elements can be accessed using the indexes store in AVL tree nodes.
3. A task ID in an AVL node must have an index associated to a heap element that contains (1) the task's designated priority number, and (2) a pointer pointing to the AVL node itself.

Besides the public operations defined in PQdeclare.h, there are 1 added public function and 19 added private functions.

For debugging purposes, we added one public function called *print_all()*. It prints, if not empty:

1. (1) The current size of the heap array
2. (2) The task ID with the smallest priority
3. (3) All nodes in the AVL tree and their task IDs and heap indexes (performed by *printTree()*)
4. (4) The visualization of the tree with the task IDs, the heap indexes, and the addresses of all nodes (performed by *displayLinks()*)
5. (5) All pairs in the binary min-heap array with their indexes, priorities, and pointers of AVL nodes

We also added the following 17+2 internal methods for AVL tree constructions and operations and Binary Heap constructions and operations (<u>mostly from part of the textbook author's heap or AVL tree classes</u>):

- (1) Two tree insertions with slight modifications
    - a. Now takes 3 inputs: Task ID, Subtree Root, and Priority
- (2) One tree node removal with slight modifications
    - a. Now swap both the Task ID and the Priority (with its successor)
- (3) One tree balance, four tree rotations, and two supplementary functions *height()* and *max()*
- (4) Three node searches (i.e., find min, find max, find which contains a given Task ID)
    - a. The third function called *findNode()* is new. It searches and returns the address of a node containing the Task ID intended to search. It returns a null pointer if not found.
- (5) One tree *makeEmpty()*
- (6) Two tree displaying with *displayLinks()* from the previous assignment
    - a. Now additionally prints indexes of nodes
- (7) One tree traversal called *traverseTreeIndex()* is newly designed for *deleteMin()* to update pointer addresses in all pairs in the heap array:
    - a. For each node traversed, get its index
    - b. For each index, go to the pair at that index in the heap array
    - c. For each pair reached, set the pointer at the second element to point to the node just traversed
- (8) One heap building
- (9) One heap node percolating down

Using the above accessing methods and functions, we can create, update, and remove a task:

1. Create:
   a. Constructor (with given sets of tasks and priorities)
      i. Insert all task IDs using AVL nodes while leaving the indexes blank
      ii. Use *findNode()* to find node addresses for all task IDs
      iii. Pair up these addresses (pointers) with their corresponding priorities
      iv. Insert these pairs into a heap array and build heap
      v. For each pair in the heap, use the pointer to update the index for each AVL node
   b. Insertion
      i. Insert the task ID using an AVL node while leaving the index blank
      ii. Find the address for the inserted task ID after the tree's balanced
      iii. Pair the address (pointer) with the priority and insert to the heap
      iv. After percolating up the heap, use the pointer to update the index for the corresponding AVL node
2. Update Priority
   a. Find the address of the AVL node containing the task to be updated
   b. Use the index stored in the node to access its heap element
   c. Make a copy of the old priority in the heap element
   d. Change the priority to the new one
   e. Percolate up/down if needed
   f. Use the pointer stored in each heap element to update the index for the corresponding AVL node
3. Delete min
   a. Store the old min into the private container *recentlyDeleted*
   b. Overwrite the top heap element with the last
   c. Reduce current size of the heap array and percolate down
   d. Use the pointer stored in each heap element to update the index for the corresponding AVL node
   e. Remove the AVL node containing the task to be deleted and balance the tree
   f. Use the index in each AVL node to update the pointer address in each heap element (using *traverseTreeIndex()*)
   g. Return the old min stored at the beginning