Name: Ziwei Zhou
UFID: 9561-9492
UF Email: ziweizhou@ufl.edu

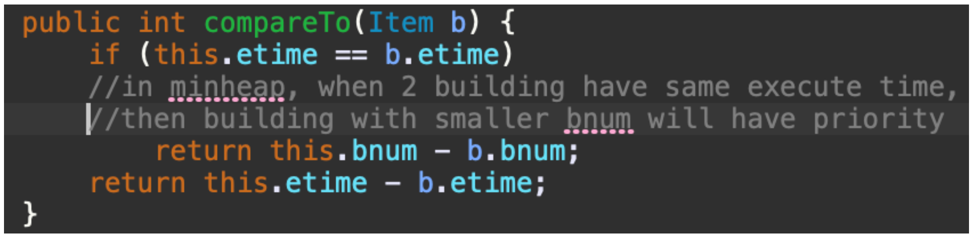
Rising City

--Report of ADS project

In this project, I implement Minheap and Red-Black Tree as my data structure to achieve the task. Details are as follows.

(1) Minheap(is sorted by execute time):

I implemented array to save data information, and extends the elements to when two building' s execute time are equal, then building has smaller bnum has higher priority.

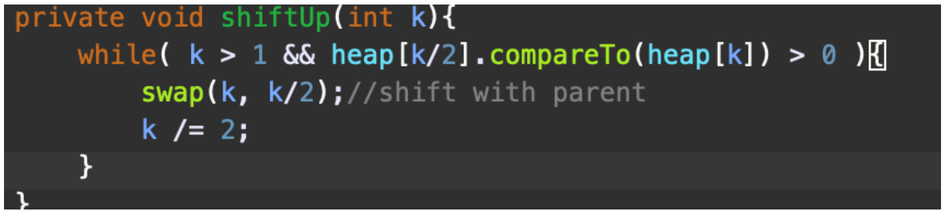


```
public int compareTo(Item b) {
    if (this.etime == b.etime)
    //in minheap, when 2 building have same execute time,
    //then building with smaller bnum will have priority
        return this.bnum - b.bnum;
    return this.etime - b.etime;
}
```

As minheap is a complete binary tree, so tree[i]' s parent is tree[i/2]; tree[i]' s left child is tree[i*2]; tree[i]' s right child is tree[i*2+1].

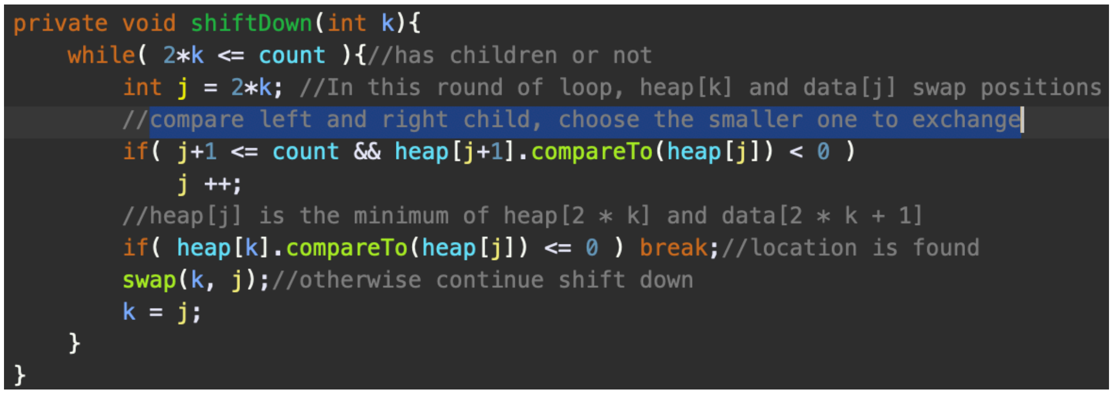
Two main method:

1. ShiftUp function: apply when data is inserted to the end of the array and by swap with its parents to find its proper location by shiftUP



```
private void shiftUp(int k){
    while( k > 1 && heap[k/2].compareTo(heap[k]) > 0 ){
        swap(k, k/2);//shift with parent
        k /= 2;
    }
}
```

2. ShiftDown function: apply after extrating top element, element at the end of array will fill vacancy, and swap with its children by shiftDown. And every time compare left and right child, choose the smaller one to exchange



```
private void shiftDown(int k){
    while( 2*k <= count ){//has children or not
        int j = 2*k; //In this round of loop, heap[k] and data[j] swap positions
        //compare left and right child, choose the smaller one to exchange
        if( j+1 <= count && heap[j+1].compareTo(heap[j]) < 0 )
            j ++;
        //heap[j] is the minimum of heap[2 * k] and data[2 * k + 1]
        if( heap[k].compareTo(heap[j]) <= 0 ) break;//location is found
        swap(k, j);//otherwise continue shift down
        k = j;
    }
}
```

For minheap, it’ s supposed to be used in work routine, which means the execute time of any building would change as global time is changing, and while working on one building, it should be extracted from minheap, and after it should be inserted back if building has not been finished.

(2) RBT(is sorted by building number):
I used node to save the information of each building, including building number, execute time and total time. Constructing a RBT to save the tree’ s information, when there’ s a print command, print node from RBT in order which cost less time compare with extract minimum for n time from minheap.
To constructing node:
1. Two basic operation for fix the balance of RBT
1.1. LeftRotate:

```
         px                                  px
        /                                   /
       x                                   y
      /  \        --(left rotate x)-.     / \
    lx    y                              x    ry
         /   \                          /  \
       ly     ry                      lx     ly
```

1.2. RightRotate:

```
          py                                  py
          /                                   /
         y                                   x
       /  \        --(right rotate x)-.     /  \
      x    ry                             lx    y
     / \                                       / \
   lx    rx                                  rx   ry
```

2. two function helper
2.1. FixInsert: After inserting node, RBT will be out of balance
    If parent x exist and is Red, continue fix upward
      a. If parent is grandparent's left child:
         a1. uncle is red :
               set uncle and parent as black, set grandparent as red
         a2. uncle is black, and "node" is right child:
               left rotate parent of x, assign the parent of x to x
         a3. uncle is black, and "node" is left child
               set parent as black, set grandparent as red, right rotate grandparent
      b. If parent is grandparent's right child:
         b1. uncle is red:

set uncle and parent as black, set grandparent as red

b2. uncle is black, and "node" is left child

right rotate parent of x, assign the parent of x to x

b3. uncle is black, and "node" is right child

set parent as black, set grandparent as red, left rotate grandparent

2.2. FixDelete: After deleting node, RBT will be out of balance

This function is implemented when a black node is deleted because it violates the black depth property of RBT.

The extra black can be removed if:

a. It reaches the root node.
b. If x points to a red-black node. In this case, x is colored black.
c. Suitable rotations and recolorings are performed.

To retain RBT: continue fix until parent x is not the root of the tree and the color of x is BLACK

a. if x is the left child of its parent
    - a1. Assign sibling to the sibling of x
    - a2. If the sibling of x is red
        - a21. Set the color of the right child of the parent of x as black
        - a22. Set the color of the parent of x as red
        - a23. Left rotate parent of x
        - a24. Assign the right child of the parent of x to sibling
    - a3. If the color of both the right and the left child of w is black
        - a31. Set the color of sibling as red
        - a32. Assign the parent of x to x
    - a4. Else if the color of the right child of sibling is black
        - a41. Set the color of the left child of sibling as black
        - a42. Set the color of sibling as red.
        - a43. Right rotate sibling
        - a44. Assign the right child of the parent of x to w
    - a5. Else:
        - a51. Set the color of sibling as the color of the parent of sibling
        - a52. Set the color of the parent of parent of x as black
        - a53. Set the color of the right child of w as black
        - a54. Left rotate the parent of x
        - a55. Set x as the root of the tree
b. if x is the right child of its parent (similar to (a))
c. set the color of x as black

3. two main function

3.1. Insert:

a. Consider RBT as a binary search tree and add nodes to the binary search tree

b. set node as red

c. fix the balance

3.2. Delete(x):

a. save the color of x in color0

b. if the left child of x is null

b1. Assign the right child of x to rx

b2. Translate x with rx

c. else if the right child of x is null

c1. Assign the left child of x into lx

c2. Transplant x with lx

d. else

d1. Assign the minimum of right subtree of x into y

d2. Save the color of y in color0

d3. Assign the right child of y into x

d4. If y is a child of x, then set the parent of x as y

d5. Else transplant y with right child of y

d6. Transplant x with y

d7. Set the color of y with color0

e. if color0 is black, use FixDelete(x) to fix the balance

(3) Main:

1. Read the input.txt file by line, and use four character :(,) to split each line, after split, each part means the date when giving command, the operation need to be done, and building number. Some line include four parts, some include three, when it’ s 3, add a “null” to make it four part to make it easy to process afterwards.
2. Start the building procedure:

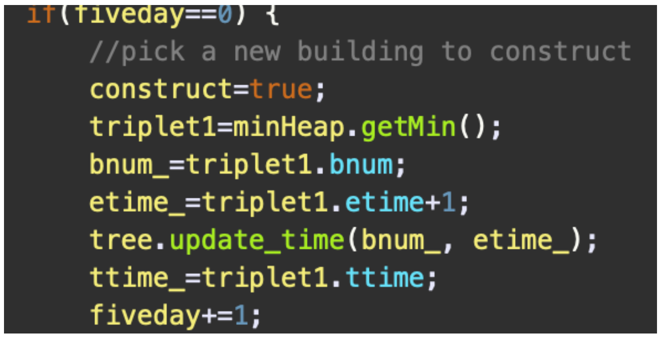
   In the outer cycle, use a global_time to count the date.

   And every day, there is a work routine where company need to work on a building, make its execute time plus 1; and if the date of command matches the global_time, need to do the operation in the input.txt.

   2.1. Work routine: add a variable(fiveday) to make a building to be constructed under 5 days:

   While there is building in minheap:

   a. fiveday==0, extract top building from minheap



```
if(fiveday==0) {
    //pick a new building to construct
    construct=true;
    triplet1=minHeap.getMin();
    bnum_=triplet1.bnum;
    etime_=triplet1.etime+1;
    tree.update_time(bnum_, etime_);
    ttime_=triplet1.ttime;
    fiveday+=1;
```

   b. fiveday!=0,

   b1. While fiveday<5 (one building need to be work for continuously 5 days when not finished)

b1.1. when(execute time<total time)building not finished:

etime++; update execute time of building in RBT, insert it back after fivedays

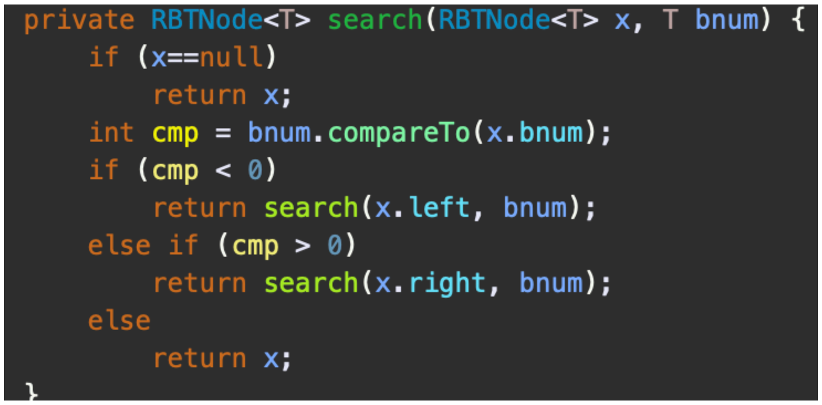
b1.2. when building finishes:

remove this building from RBT and minheap(as for minheap, since it has been already extracted, just no need to insert back to minheap.

2.2. Command day matches:

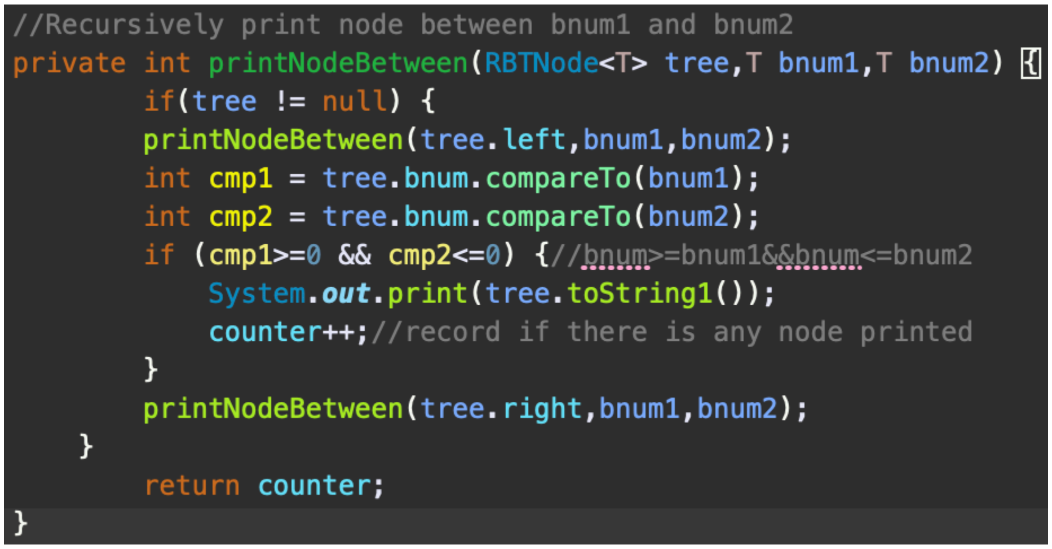
Three kinds of operations:

a. Insert: insert (bnum, execute time=0, total time) into minheap and RBT
b. PrintBuilding(x): recursively search node x in RBT, and after node is found, print, otherwise, print(0, 0, 0)



```
private RBTNode<T> search(RBTNode<T> x, T bnum) {
    if (x==null)
        return x;
    int cmp = bnum.compareTo(x.bnum);
    if (cmp < 0)
        return search(x.left, bnum);
    else if (cmp > 0)
        return search(x.right, bnum);
    else
        return x;
}
```

c. PrintBuilding(x, y): use a counter as a global variable and count when there happens print one building, if counter==0, which means there is no building satisfy the condition(x<bnum<y), then print(0, 0, 0)



```
//Recursively print node between bnum1 and bnum2
private int printNodeBetween(RBTNode<T> tree,T bnum1,T bnum2) {
        if(tree != null) {
        printNodeBetween(tree.left,bnum1,bnum2);
        int cmp1 = tree.bnum.compareTo(bnum1);
        int cmp2 = tree.bnum.compareTo(bnum2);
        if (cmp1>=0 && cmp2<=0) {//bnum>=bnum1&&bnum<=bnum2
            System.out.print(tree.toString1());
            counter++;//record if there is any node printed
        }
        printNodeBetween(tree.right,bnum1,bnum2);
    }
        return counter;
}
```

2.3. Write result in output.txt