BPlusTree：

Part1 主体部分

①BPlusTree设计：

1. 节点：分为叶子节点和内部节点，叶子节点中有pair<Key, Data>[Max\_data\_num]用以存数据；内部节点中有keys[Max\_key\_num]和childs[Max\_child\_num]用来存关键字和子节点的位置。两种节点均有代表节点类型的Node\_type，代表父亲节点位置的指针par和代表关键字数量的key\_num。
2. 树：树中有一个root指针指向头结点，num\_of\_data代表数据存储量。

②函数设计：

1. insert函数：在数据量为0时，为root分配空间，并把它的类型设置为Leaf；数据量不为0时，从root向下找合适的child直至child的类型为Leaf，其中，对于遇到的每一个中间节点，若其大小大于Max\_key\_num，则对它调用split函数。找到对应位置的Leaf节点后插入，若其key\_num >= Max\_data\_num，对它调用split函数。
2. split函数：将这个节点均分成两半。
3. remove函数：对于root的key\_num为0或1，或root的node\_type为Leaf时特殊考虑。自顶向下寻找合适的child，其中，若节点的key\_num <= Max\_key\_num时对它调用adjust函数。找到对应位置的Leaf节点后进行删除，若其key\_num < Min\_data\_num，则对它调用adjust函数。
4. adjust函数：
   1. 若其右节点为空或与它不是同一个父亲：

if(左节点的key\_num + 它的key\_num <= Max\_key\_num 或 Max\_data\_num)

根据它的类型对它调用left\_internal\_merge函数或left\_leaf\_merge函数

else

根据它的类型对它调用borrow\_left\_internal\_node函数或borrow\_left\_leaf\_node函数

* 1. 否则

if(右节点的key\_num + 它的key\_num <= Max\_key\_num 或 Max\_data\_num)

根据它的类型对它调用right\_internal\_merge函数或right\_leaf\_merge函数

else

根据它的类型对它调用borrow\_right\_internal\_node函数或borrow\_right\_leaf\_node函数

1. clear函数：通过递归实现。
2. find函数和[]运算符重载：[]重载函数若寻找的关键字不存在，则抛出out\_of\_index()。

Part2 文件操作设计

对于一棵树，共需要三个文档存储，一个文档存中间结点的信息，一个文档存叶子节点（即所有数据）的信息，一个文档存树的信息（包括树的大小，root节点的类型和位置）。在构建BPlusTree时，类的使用者传入三个string类变量作为文档名称，若文档不存在，则创建，否则打开文档。打开的方式为二进制文件。

节点的设计与上面类似，不过此时，root中存储的是一个long型的变量，代表root节点的位置在对应文档的哪里。所有child指针和par指针也是如此。此外，还设置了一个mypos用来存储该节点自己的位置。

每次函数需要用到节点时，通过节点的位置找到对应文件里的数据，用一个临时节点暂存，对它进行操作，操作完后又存回去。

在删除时，为了防止删除过多文档太大，在该树中以一种链表的形式存储被删除节点的位置。每个文档最开始留下空间，存储数据为0。当删除一个节点时，让文档最开始的地方存储这个节点的位置，这个节点的地方存储原先文档最开始的地方存储的位置，形成一条链表。在需要添加节点时，若文档最开始处存储的数据是0，则在文档末尾添加，否则取出链表的第一个节点的位置用来放置新节点。

Class: User ——每个用户的信息

·pas：密码

·name：用户名

·mail：邮箱地址

·pri：所处用户组优先级

每个用户下有一个按时间顺序的链表，记录此用户的所有订单。

·fir：订单链表的头结点

·cur：订单链表的尾结点

·cnt：订单数量

·isLog：是否已登录

Class: Train ——每个车次的信息

·num：车次经过的车站数量

·pri[100]：从起点到每一站的累计票价

每个车次每一天在每一站的剩余票数记在了res数组里

·r[100]：对应res数组中每天在起点的剩余票数的位置，接下来每一站的剩余票数都是按顺序存的

·p[N]：

·dm[2][N]：从起点到达每一站/从每一站出发与从起点发车时的天数差

·st：起点站发车的时间

·ar[N]：每一站的到达时间

·lv[N]：每一站的出发时间

每个车次的每一天下都有一个按时间顺序的链表，记录订单。

·fir[N]：每一天的订单链表头结点

·cur[N]：每一天的订单链表尾结点

·sta[100]：车站的名字

·ty：列车类型

·S，T：车次的售卖时间区间

·isr：是否发布

Class: data ——每个车站的信息

每个车站下有一个链表1记录经过这个车站的所有车次

·fir：车次链表的头结点

·cur：车次链表的尾结点

每次query\_transfer重新搜集的信息，用链表2记录从当前车站到达询问的终点站有哪些车次

·fir\_：该链表的头结点

·cur\_：该链表的尾结点

用三棵BPlusTree存<用户ID, User>的映射user，<火车ID, Train>的映射train，<车站名, data>的映射sta。

操作

add\_user()：

在user中插入用户信息

Login()：

用户登录，将islog改为1

Logout()：

用户登录，修改islog为0

query\_profile()：

在user中查询用户信息

modify\_profile()：

在user中修改用户信息

add\_train()：

在train中插入火车信息；更新sta的链表1信息，在经过的每一站的链表1中加入这个车次

release\_train()：

火车发布，将isr改为1

query\_train()：

在train中查询车次信息

delete\_train()：

在train中删除该车次；修改sta的链表1信息，遍历经过的每一站的链表1，删除该车次

query\_ticket()：

遍历sta[起点]的链表1，将经过终点的车次存下来，最后排序

query\_transfer()：

思路：用sta[起点]和sta[终点]找出从起点到每个可能的中转站，以及中转站到终点的信息。在中转站合并，找出最优答案。

遍历sta[终点]的链表1，将每个车次加到sta[经过的每一站]（可能的中转站）的链表2中。再遍历sta[起点]的链表1，每个车次经过的每一站都是可能的中转站。遍历sta[该车站]的链表2，算出答案。

buy\_ticket()：

不失败的情况下在user的链表中插入该订单

若立刻成功，则修改train中的剩余票数信息；若候补，则加入该车次和日期下的候补链表

query\_order()：

遍历user中的链表，输出订单

refund\_ticket()：

修改train中的剩余票数信息；遍历该车次和日期下的候补链表，将可买的订单买了，从候补链表中移除

clean()：

清除数据

exit():

将所有在线用户logout()