B+tree（内存排序+ bufferpool+读写优化+父亲链）

B+树结构说明:

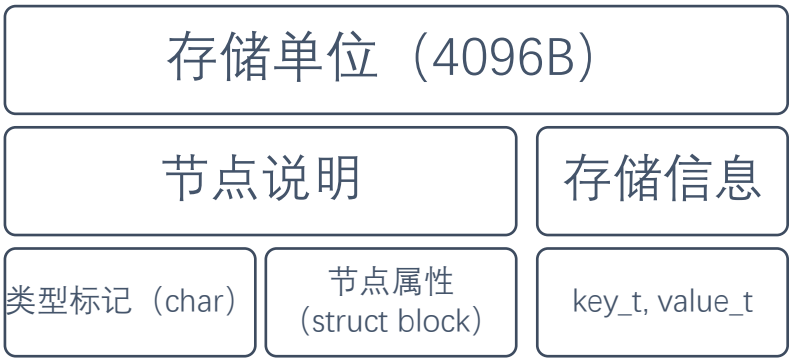
单文件存储：

利用 blank_begin 和 blank_end 分别记录文件的第一块可用空间和文件末尾，对于每一块删除节点后产生的剩余空间，采用类似邻接表结构进行维护。

内部节点：

内部节点的节点属性记录节点容量、所在文件位置，不记录父节点。

叶节点：



叶节点的节点属性记录节点容量、所在文件位置、下一叶节点所在文件位置，不记录父节点。

B+树优化策略：

Bufferpool:

采用哈希表在内存中保存一些最近访问过的节点，减少外存访问次数。

内存排序:

考虑到实际的建树操作过程并非有序，Bufferpool 的优化能力大大下降，为了保证相邻两次插入过程需要访问的节点大致相同，考虑在内存中开辟一块区域先在内存中进行排序再有序插入的策略。

读写优化：

将 B+树的节点大小设置为 4096B，充分利用 fread 以及 fwrite 的缓冲区域。

父亲链：

B+树节点不保存父节点，在每次访问叶节点时采用数组维护父子关系。

B+树接口：

构造函数

`bptree(const char * fname)`

fname:储存 B+树数据的文件

- 删库

`init()`

- 根据 key 查找元素

`value_t find(const key_t &key, const value_t & d = value_t())`

返回 key 所对应的 value，若 key 不存在，返回 d

- 插入元素

`insert(const key_t &key, const value_t &v)`

若 key 已存在，则什么事情都不会做

- 修改

`void set(const key_t &key, const value_t &v)`

如果不存在 key，会报错，返回 not_found

- 删除

`void remove(const key_t &key)`

如果不存在 key，会报错，返回 not_found

- 区间查找

`void search(vector & arr, const key_t & key, std::function<bool(const key_t &, const key_t &)> compar)`

compar 的意义为“等于”，

后端架构