자료구조론

B Tree

u@hataeseong-ui-MacBook-Pro:~/Desktop/2017_CSE2010_2016025041/HW7$./a.out input.txt
key: 50
key: 58
key: 59
key: 60
u@hataeseong-ui-MacBook-Pro:~/Desktop/2017_CSE2010_2016025041/HW7$./a.out input2.txt
key: 5
key: 6
key: 7
key: 10
key: 12
key: 17
key: 20
key: 30
u@hataeseong-ui-MacBook-Pro:~/Desktop/2017_CSE2010_2016025041/HW7$
- 실행결과 일치

```
/* changes the shape of the tree after btInsertInternal operation */
void rearrange(bTree b, bTree b2,int pos, int mid){
  memmove(&(b->keysAndKids.keys[pos + 1]), &(b->keysAndKids.keys[pos]),
                                sizeof(*(b->keysAndKids.keys)) * (b->keysAndKids.numKeys - pos));
   memmove(&(b->keysAndKids.kids[pos + 2]), &(b->keysAndKids.kids[pos]),
                                sizeof(*(b->keysAndKids.keys)) * (b->keysAndKids.numKeys - pos));
   b->keysAndKids.keys[pos] = mid;
   b->keysAndKids.kids[pos + 1] = b2;
   b->keysAndKids.numKeys++;
}
```

- pos 이후의 키를 하나씩 옮겨 자리를 만들어 mid를 넣고 pos 이후의 자식을 옮겨 pos+1에 새 트리 연결

```
static bTree btInsertInternal(bTree b, int key, int *median)
{
   int pos;
   int mid;
   bTree b2;

   pos = searchKey(b->keysAndKids.numKeys, b->keysAndKids.keys, key);

   if(pos < b->keysAndKids.numKeys && b->keysAndKids.keys[pos] == key) {
      /* nothing to do */
      return 0;
   }

   if(b->isLeaf) {
      memmove(&(b->keysAndKids.keys[pos + 1]), &(b->keysAndKids.keys[pos]),
                                    sizeof(*(b->keysAndKids.keys)) * (b->keysAndKids.numKeys - pos));
      b->keysAndKids.keys[pos] = key;
      b->keysAndKids.numKeys++;
   }

   else {
      /* insert in child */
      b2 = btInsertInternal(b->keysAndKids.kids[pos], key, &mid);
       /* insert a new key in b */
      if(b2) {
            rearrange(b,b2,pos,mid);
      }
   }
   /* we waste a tiny bit of space by splitting now
    * instead of on next insert */
   if(b->keysAndKids.numKeys >= MAX_KEYS) {
      mid = b->keysAndKids.numKeys/2;
      *median = b->keysAndKids.keys[mid];

      b2 = (bTree)malloc(sizeof(*b2));

      b2->keysAndKids.numKeys = b->keysAndKids.numKeys - mid - 1;
      b2->isLeaf = b->isLeaf;

            memmove(b2->keysAndKids.keys, &b->keysAndKids.keys[mid+1]
                        , sizeof(*(b->keysAndKids.keys)) * b2->keysAndKids.numKeys);
     if(!b->isLeaf) {
```

```
            memmove(b2->keysAndKids.kids, &b->keysAndKids.kids[mid+1]
                                    , sizeof(*(b->keysAndKids.kids)) * (b2->keysAndKids.numKeys + 1));
        }

        b->keysAndKids.numKeys = mid;

        return b2;
    }
    else {
        return 0;
    }
}
```

- 새로운 키가 들어갈 위치를 search함수로 찾아서 그 위치가 리프트리이면 바로 삽입 후 함수 종료, 리프트리가 아닐 경우 그 노드를 기준으로 다시 재귀형식으로 실행 후 rearrange하고 만약 MAX_KEY보다 키가 많을 경우 새로운 리프트리를 만들어서 리턴

```
void btInsert(bTree b, int key)
{
    bTree b1;   /* new left child */
    bTree b2;   /* new right child */
    int median;

    b2 = btInsertInternal(b, key, &median);

    if(b2) {
        /* basic issue here is that we are at the root */
        /* so if we split, we have to make a new root */
        b1 = (bTree)malloc(sizeof(*b));
        assert(b1);

        /* copy root to b1 */
        memmove(b1, b, sizeof(*b));

        /* make root point to b1 and b2 */
        b->isLeaf = 0;
        b->keysAndKids.numKeys = 1;
        b->keysAndKids.keys[0] = median;
        b->keysAndKids.kids[0] = b1;
        b->keysAndKids.kids[1] = b2;
    }
}
```

- b 트리 내부에 key를 넣는 btInsertInternal함수 실행하고 만약 btInsertInternal 함수에서 b2가 리턴된 경우(새로운 리프트리를 만든 경우)에는 거기에 해당되는 루프트리를 만들어서 b2와 연결