Assignment #2

B+tree Practice

DBMS
2021045796
Kimdogyeom
김도겸

1. 코드 분석

struct record	key값과 char type values를 갖는다.
struct inter_record	key값과 pointer offset 값을 갖는다.
struct Page	본인이 leaf인지 확인한다.
struct Header_Page	header page 역할을 수행한다.
int fd	file direction으로 추정
page *rt	root
H_P *hp	header page
int open_table(char *pathname)	path name을 입력받아 table을 연다
H_P *load_header(off_t off)	table의 header를 가져온다
page *load_page(off_t off)	page를 가져온다
void reset(off_t off)	reset이라는 임시 page를 제작한다. reset의 정보를 삭제하고, free(reset)을 해준다.
off_t new_page()	새로운 페이지를 제작하여 header의 끝에 연결한다.
off_t find_leaf(int64_t key)	leaf의 어디에 연결되어 있는지를 찾는다.
char * db_find(int64_t key)	char *value를 지정한다. find_leaf를 호출하여 위치를 탐색한다. 0(없음)을 return 받은 경우에는 함수를 종료한다.
void freetouse(off_t fpo)	reset이라는 임시 page를 제작하고, load_page를 호출하여 reset page를 삭제한다.
int cut(int length)	내부 노드의 길이가 제한 범위를 넘었을 때, 반으로 쪼개 2개의 노드로 바꾸어준다.
int parser()	사용되지 않는 듯 하다.
void start_new_file(record rec)	root page를 제작하여 새로운 file을 제작해준다.
int db_insert(int64_t key, char * value)	root가 null이면 start_new_file을 호출, db_find한 결과가 null이 아니라면 free, 이후 insert_into_leaf를 해주는데, 왜 조건이 붙어있는지 잘 모르겠다. bpt.c 231번째 줄, 분석 필요
off_t insert_into_leaf(off_t leaf, record inst)	load_page를 호출하여 page를 지정하고, pwrite 함수를 호출한다.
off_t insert_into_leaf_as(off_t leaf, record inst)	page에 성공적으로 node를 삽입하였는지 확인하는 듯하다. 반복되는 과정이 존재하기 때문에 수정했을 때 속도가 올라갈 가능성이 보이는 함수이다.
off_t insert_into_parent(off_t old, int64_t key, off_t newp)	부모 노드에 어떻게 삽입할지를 결정하는 함수인 듯하다. free와 insert_into_internal_as가 조건이 붙어서 2번 반복되는 이유가 무엇인지 확인해보아야 할 것 같다.

int get_left_index(off_t left)	return number of left node index
off_t insert_into_new_root(off_t old, int64_t key, off_t newp)	new page를 생성하고, load한다. left page & right page를 old page와 newp page로 연결한다.
off_t insert_into_internal(off_t bumo, int left_index, int64_t key, off_t newp)	부모의 page를 가져와서 값을 삽입하는 듯하다.
off_t insert_into_internal_as(off_t bumo, int left_index, int64_t key, off_t newp)	삽입 과정에서 일어나고 있는 일을 정리하는 듯하다. 새로운 page를 제작하여 insert 과정에서 split이 필요한 경우 트리를 수정한다. bpt.c 428번 줄에 있으며, overhead를 줄일 여지가 보인다.
int db_delete(int64_t key)	db_find 이후 delete 할 node를 찾고, find_leaf를 거쳐 값을 삭제한다.
void delete_entry(int64_t key, off_t deloff)	remove_entry_from_page를 호출한다. delete 한 이후 조건에 맞지 않는 page를 수정한다. 크기가 max를 넘었는지 확인한 후 각각의 상황에 따라 coalesce_pages와 redistribute_pages를 호출한다.
void redistribute_pages(off_t need_more, int nbor_index, off_t nbor_off, off_t par_off, int64_t k_prime, int k_prime_index)	redistributable: 재배포 가능 ->redistribute_pages: 재배포하는 페이지 leaf인지 아닌지를 판별하고, leaf인 경우에는 값을 수정하는 듯하다.
void coalesce_pages(off_t will_be_coal, int nbor_index, off_t nbor_off, off_t par_off, int64_t k_prime)	coalesce: 합쳐지다 coalesce_pages: 두 page를 합친다 일정 조건을 만족했을 때 한 page의 값을 이동시키고, 다른 page의 entry를 삭제한다.
void adjust_root(off_t deloff)	root node의 조건에 문제가 생겼을 때, 수정한다
void remove_entry_from_page(int64_t key, off_t deloff)	page로부터 entry를 제거한다. 삭제할 page를 load하고, 조건에 맞추어 tree를 조정한 후에 free한다.
void usetofree(off_t wbf)	정의되어있지 않다.

2. 수정 계획

delete	고스트 정리를 이용하여 삭제 과정에서 발생하는 overhead를 최소화한다.
db_insert	insert과정에서 조건이 붙지만 같은 일을 반복하는 경우에 원인을 파악하여 함수의 선언 횟수를 줄일 수 있는지 확인해본다.
insert_into_leaf_as	반복되는 부분을 정리하여 DP에 저장하는 방식으로 overhead를 줄일 수 있을 듯하다.
db_delete	db_find를 통해 필요한 key를 find_leaf 해주는데, 이후에 값을 찾고 난 이후에도 find_leaf를 반복해준다. 이 과정에서 중복되는 함수를 제거하여 overhead를 줄인다.
insert_into_internal_as	조건이 붙지만 같은 일을 반복하는 경우에 원인을 파악하여 함수 선언 횟수를 줄일 수 있는지 확인해본다.

3. 테스트 케이스 정보

```
#include"bpt.h"
#include<time.h>

intmain(){
   int64_tinput;
   charinstruction;
   charbuf[120];
   char*result;
   open_table("test.db");
   inti=0;

   intj,k=1001;

   clock_tstartTick,endTick;
   structtimevaltv;
   while(scanf("%c",&instruction) !=EOF){

     switch(instruction){
       case'i':
         startTick=clock();
         scanf("%ld%s",&input,buf);
         db_insert(input,buf);
         endTick=clock(); //
         printf("%d> Tick: %d\n",++i,endTick-startTick);//
```

```
        break;
      case'f':
        scanf("%ld",&input);
        result=db_find(input);
        if (result) {
          printf("Key: %ld, Value: %s\n",input,result);
        }
        else
          printf("Not Exists\n");

        fflush(stdout);
        break;
      case'd':
        scanf("%ld",&input);
        db_delete(input);
        break;
      case'q':
        while (getchar() != (int)'\n');
        returnEXIT_SUCCESS;
        break;

      case'm':
        startTick=time(NULL);
        for(j=1; j<k; j++) {
          sprintf(buf,"%d",k-j);
          db_insert(j,buf);
          printf("%d Done!\n",j);
        }
        endTick=time(NULL); //
        printf("%d> Tick: %d\n",++i,endTick-startTick);//
        break;
      case'a':
        startTick=time(NULL);
        for(j=1; j<k; j++) {
          //sprintf(buf, "%d", k-j);
          db_delete(j);
          printf("%d Done!\n",j);
        }
        endTick=time(NULL); //
        printf("%d> Tick: %d\n",++i,endTick-startTick);//
```

```
        break;
      case'k':
        gettimeofday(&tv,NULL);
        scanf("%d",&k);
        doublestart2= (tv.tv_sec) *1000+ (tv.tv_usec) /1000;
        for(j=1; j<k; j++) {
          sprintf(buf,"%d",k-j);
          db_insert(j,buf);
          //printf("%d Done!\n",j);
        }
        for(j=1; j<k; j++) {
          //sprintf(buf, "%d", k-j);
          db_delete(j);
          //printf("%d Done!\n",j);
        }
        gettimeofday(&tv,NULL);
        doubleend2= (tv.tv_sec) *1000+ (tv.tv_usec) /1000;
        printf("%d> Time: %f\n",++i, (end2-start2)/1000);//


    }

    while (getchar() != (int)'\n');
  }
  printf("\n");
  return0;
}
```

main.c 파일입니다.
테스트는 case ‘k’를 이용하여 자료의 삽입과 삭제를 진행합니다.

‘k 10001’

과 같은 방법으로 실행합니다.(이 경우에는 10000개의 자료의 삽입과 삭제를 진행합니다.)

최종 결과본과는 다른 실험용 자료이며, main.c를 수정할 경우 이를 기반으로 수정하여 시간을 측정하였습니다.

4. Trouble Shooting

1) 선형 탐색 방식 수정하기

db_find 함수는 자료의 위치를 찾는데 있어서 선형 탐색 방식을 사용합니다. 선형 탐색 방식을 이진 탐색 방식으로 수정하여 보다 빠른 탐색을 목표로 하였습니다.

```
char*db_find(int64_tkey) {
  char* value = (char*)malloc(sizeof(char) *120);
  int i =0;
  off_t fin =find_leaf(key);
  if (fin ==0) {
   returnNULL;
  }
  page * p =load_page(fin);

  for (; i <p->num_of_keys; i++) {
   if (p->records[i].key== key) break;
  }
  if (i ==p->num_of_keys) {
   free(p);
   returnNULL;
  }
  else {
   strcpy(value,p->records[i].value);
   free(p);
   return value;
  }
}
```

기존 db_find, 선형 탐색 방식의 반복문을 사용합니다.

```
char*db_find(int64_tkey) {
  char*value= (char*)malloc(sizeof(char) *120);
  inti=-1;
  off_tfin=find_leaf(key);
  if (fin==0) {
   returnNULL;
  }
  page*p=load_page(fin);
  //* dogyeom
  intlow=0;
  inthigh=p->num_of_keys-1;
  intmid;
  while(low<=high) {
   mid= (low+high) /2;
   if(p->records[mid].key==key) {
    i=mid;
```

```
      break;
    }
    elseif(p->records[mid].key>key) {
      high=mid-1;
    }
    else {
      low=mid+1;
    }
  }
  if (i==-1) {
    free(p);
    returnNULL;
  }
  else {
    strcpy(value,p->records[i].value);
    free(p);
    returnvalue;
  }
}
```

수정한 db_find, 이진 탐색 방식을 사용합니다.

좌측이 이진 탐색을, 우측이 기존 선형 탐색을 사용한 버전의 b+tree입니다. 자료 300개를 삽입하고 삭제한 결과, 선형 탐색 방식이 더 빨랐습니다.

자료 30000개를 삽입했을 때의 결과입니다. 선형 탐색 방식이 이진 탐색에 비해 약 50초 정도 더 빨랐습니다.

다만, 자료 300000개를 삽입했을 때는 시간이 거의 유사했습니다. 30만개를 기준으로 선형 탐색과 이진 탐색을 나누어 탐색하도록 한다면 효율을 높일 수 있을 것이라 생각합니다.

자료 400000개를 삽입했을 때도 선형 탐색 방식이 빨랐지만, 500000개를 삽입하면서 부터는 이진 탐색 방식이 더 빨라지기 시작했습니다. 40~50사이의 확실한 기준점을 찾으려 했지만, 다른 방식의 시간 차이의 비교를 위하여 tree에 삽입된 자료의 크기가 500000 이상일 때부터 이진 탐색을 진행하는 방식으로 수정하고자 합니다.

```
char*db_find(int64_tkey) {
  char*value= (char*)malloc(sizeof(char) *120);
  inti=-1;
  off_tfin=find_leaf(key);
  if (fin==0) {
   returnNULL;
  }
  page*p=load_page(fin);
  if(p->num_of_keys>=500000) {
   //* dogyeom
   intlow=0;
   inthigh=p->num_of_keys-1;
   intmid;
   while(low<=high) {
    mid= (low+high) /2;
    if(p->records[mid].key==key) {
     i=mid;
     break;
    }
    elseif(p->records[mid].key>key) {
     high=mid-1;
    }
    else {
     low=mid+1;
    }
   }
   if (i==-1) {
    free(p);
    returnNULL;
   }
   else {
    strcpy(value,p->records[i].value);
    free(p);
    returnvalue;
   }
  }
```

```
    else {
      for (; i<p->num_of_keys; i++) {
        if (p->records[i].key==key) break;
      }
      if (i==p->num_of_keys) {
        free(p);
        returnNULL;
      }
      else {
        strcpy(value,p->records[i].value);
        free(p);
        returnvalue;
      }
    }
}
```

최종 수정된 db_find, tree의 num_of_keys에 기준을 두어 이진 탐색과 선형 탐색을 선택적으로 진행합니다.

좌측이 이진 탐색과 선형 탐색을 선택적으로 진행한 케이스이고, 우측이 기존 선형 탐색만을 이용하여 테스트를 진행한 케이스입니다. 데이터 1000000개를 삽입, 삭제한 결과 약 1초 정도 빨랐음을 확인 할 수 있었습니다. 다만 1초로 성능 개선이 얼마나 진행되었는지를 확인하기 어려워 구체적인 확인을 위해 3000000개의 데이터를 다시 삽입, 삭제 해주었습니다.

데이터 3000000개를 삽입하고 삭제했을 때, 이진 탐색과 선형 탐색을 선택적으로 진행한 케이스가 약 300초 빨랐음을 확인 할 수 있었습니다.

이러한 이유가 나온 이유에 대해 고민한 결과 선형 탐색의 경우에는 disk를 순회하는데 있어서 io가 많이 발생하지 않습니다. 하지만 이진 탐색의 경우에는 disk를 왕복하며 순회하므로 io가 많이 발생합니다. 하지만 트리의 자료 수가 일정 수치를 넘었을 때부터는 이진 탐색이 오히려 disk io가 더 적게 발생한다는 판단을 얻었습니다.

2) 임시 저장장치(~~Ghost Buffer~~)

버퍼에 저장되는 속도가 레코드에 저장되는 속도보다 빨라서 버퍼에 수를 먼저 저장한 후 record에 기록하는 것이 더 빠르다는 개념이 Ghost Buffer라고 잘못 알고 프로그램 설계를 진행했습니다. 그래서 아마 임시 저장 장치라 표현하는 것이 더 맞다고 생각합니다.

```
case'k':
        gettimeofday(&tv,NULL);
        scanf("%d",&k);
```

```
        doublestart2= (tv.tv_sec) *1000+ (tv.tv_usec) /1000;
        for(j=1; j<k; j++) {
          sprintf(buf,"%d",k-j);
          db_insert(j,buf);
          //printf("%d Done!\n",j);
        }
        for(j=1; j<k; j++) {
          //sprintf(buf, "%d", k-j);
          db_delete(j);
          //printf("%d Done!\n",j);
        }
        gettimeofday(&tv,NULL);
        doubleend2= (tv.tv_sec) *1000+ (tv.tv_usec) /1000;
        printf("%d> Time: %f\n",++i, (end2-start2)/1000);//
```

자료 삽입과 삭제

```
intinit_insert(int64_tkey,char*value,intchecker) {
   recordnr;
   nr.key=key;
   strcpy(nr.value,value);
   if(rt==NULL) {
     start_new_file(nr);
     return0;
   }
   char*dupcheck;
   dupcheck=db_find(key);
   if (dupcheck!=NULL) {
     free(dupcheck);
     return-1;
   }
   free(dupcheck);
   ghost[checker] =nr;
   //ghostCheck++;
   //dogyeom
}
```

ghost라는 자료를 저장하는 별도의 배열을 제작하여 임시로 자료를 저장하였다가 한 번에 insert를 진행하였습니다.

```
intdb_insert(intchecker) {
   for(intt=0; t<checker; t++) {
     off_tleaf=find_leaf(ghost[t].key);
     page*leafp=load_page(leaf);
     if (leafp->num_of_keys<LEAF_MAX) {
```

```
      insert_into_leaf(leaf,ghost[t]);
      free(leafp);
    }
    else {
      insert_into_leaf_as(leaf,ghost[t]);
      free(leafp);
      //why double free?
    }
  }
  //ghostCheck = 0;
  return0;
}
```

find, delete, quit의 연산을 할 때와 init_insert 횟수가 일정 횟수(100)을 넘었을 때 db_insert를 진행하고, 다른 때는 init_insert 함수를 이용하여 tree 내 존재 여부를 확인한 뒤 배열에 임시 저장을 진행하였습니다.

```
intinit_delete(int64_tkey,intdch) {
  if (rt->num_of_keys==0) {
    //printf("root is empty\n");
    return-1;
  }
  char*check=db_find(key);
  if (check==NULL) {
    free(check);
    //printf("There are no key to delete\n");
    return-1;
  }
  free(check);
  Dghost[dch] =key;
  //return 0;
}
```

delete도 insert와 마찬가지로 미리 배열에 저장했다가 한 번에 삭제하는 방식을 사용해보았습니다.

```
intdb_delete(intdch) {
/*
  if (rt->num_of_keys == 0) {
    //printf("root is empty\n");
    return -1;
  }
  char * check = db_find(key);
  if (check== NULL) {
    free(check);
```

```
    //printf("There are no key to delete\n");
    return -1;
  }
  free(check);*/
  for(intt=0; t<dch; t++) {
    off_tdeloff=find_leaf(Dghost[t]);
    delete_entry(Dghost[t],deloff);
  }
  return0;

}//fin
```

마찬가지로 find, insert, quit이 선언되었을 때 db_delete를 실행하여 Dghost 배열안에 있는 key를 가진 record를 삭제해주었습니다.

좌측이 임시 저장장치를 사용한 테스트고, 우측이 기본 테스트입니다. 둘 다 선형 탐색과 이진 탐색을 병행하는 find를 사용하였습니다. 40000개를 삽입하고 삭제했을 때와 600000개를 삽입하고 삭제했을 때, 오히려 임시 저장장치를 사용한 경우가 더 시간이 오래 걸렸습니다. 이러한 방법은 모든 record를 삽입한 후 한번에 tree를 재조정했을 때만 효과를 볼 수 있다는 점을 알게 되었습니다.

3) delay merge

B+tree에서 merge는 delete 연산 과정에서 발생합니다. void delete_entry에서 root일 때와 최소키 조건을 만족할 때를 제외하고는 void coalesce_pages를 통해 두 pages를 합쳐 주거나 void redistribute_pages를 이용하여 pages의 records를 재분배해줍니다. 여기서 발생하는 merge를 delay시키기 위해서는 최소키 조건을 조절하여 최소키를 만족하지 못하는 pages가 많을 때 전체 조정을 하는 방식이 효과가 있을 것 같다고 생각하였습니다.

```
voiddelete_entry(int64_tkey,off_tdeloff) {

  remove_entry_from_page(key,deloff);

  if (deloff==hp->rpo) {
    adjust_root(deloff);
    return;
  }
  page*not_enough=load_page(deloff);
  intcheck=not_enough->is_leaf?cut(LEAF_MAX) :cut(INTERNAL_MAX);
```

```
if (not_enough->num_of_keys>=check){ //* dogyeom
 free(not_enough);
 //printf("just delete\n");
 return;
} //최소키 조건

intneighbor_index,k_prime_index;
off_tneighbor_offset,parent_offset;
int64_tk_prime;
parent_offset=not_enough->parent_page_offset;
page*parent=load_page(parent_offset);
//이쪽을 고쳐야 delay-merge가 가능할듯
if (parent->next_offset==deloff) {
  neighbor_index=-2;
  neighbor_offset=parent->b_f[0].p_offset;
  k_prime=parent->b_f[0].key;
  k_prime_index=0;
}
elseif(parent->b_f[0].p_offset==deloff) {
  neighbor_index=-1;
  neighbor_offset=parent->next_offset;
  k_prime_index=0;
  k_prime=parent->b_f[0].key;
}
else {
  inti;

  for (i=0; i<=parent->num_of_keys; i++)
    if (parent->b_f[i].p_offset==deloff) break;
  neighbor_index=i-1;
  neighbor_offset=parent->b_f[i-1].p_offset;
  k_prime_index=i;
  k_prime=parent->b_f[i].key;
}

page*neighbor=load_page(neighbor_offset);
intmax=not_enough->is_leaf?LEAF_MAX:INTERNAL_MAX-1;
intwhy=neighbor->num_of_keys+not_enough->num_of_keys;
//printf("%d %d\n",why, max);
//dogyeom
```

```
  if (why<=max) {
    // free(not_enough);
    // free(parent);
    // free(neighbor);
     // coalesce_pages(deloff, neighbor_index, neighbor_offset, parent_offset,
k_prime);
  }
  else {
    free(not_enough);
    free(parent);
    free(neighbor);

  redistribute_pages(deloff,neighbor_index,neighbor_offset,parent_offset,k_prime,k
_prime_index);
  }

  return;

}
```

때문에 최소키 제약 조건을 만족하지 못할 때, 두 개의 page를 합치는 merge를 진행하지 않아보았습니다.

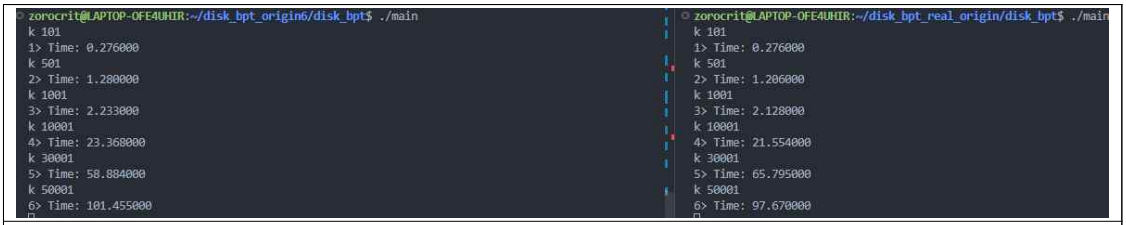
결과는 merge를 진행해준 쪽이 삽입과 삭제를 진행하는 속도가 빨랐습니다. 때문에 merge를 아예 진행하지 않는 것이 아니라 최소키 조건을 조정하는 방식으로 merge를 delay해주는 것이 효과가 있을 것이라 생각하였습니다.

```
voiddelete_entry(int64_tkey,off_tdeloff) {

  remove_entry_from_page(key,deloff);

  if (deloff==hp->rpo) {
    adjust_root(deloff);
    return;
  }
  page*not_enough=load_page(deloff);
  intcheck=not_enough->is_leaf?cut(LEAF_MAX) :cut(INTERNAL_MAX);
  if (not_enough->num_of_keys>=check*2/3){ //* dogyeom
   free(not_enough);
   //printf("just delete\n");
   return;
  } //최소키 조건
```

```
intneighbor_index,k_prime_index;
off_tneighbor_offset,parent_offset;
int64_tk_prime;
parent_offset=not_enough->parent_page_offset;
page*parent=load_page(parent_offset);
//이쪽을 고쳐야 delay-merge가 가능할듯
if (parent->next_offset==deloff) {
  neighbor_index=-2;
  neighbor_offset=parent->b_f[0].p_offset;
  k_prime=parent->b_f[0].key;
  k_prime_index=0;
}
elseif(parent->b_f[0].p_offset==deloff) {
  neighbor_index=-1;
  neighbor_offset=parent->next_offset;
  k_prime_index=0;
  k_prime=parent->b_f[0].key;
}
else {
  inti;

  for (i=0; i<=parent->num_of_keys; i++)
    if (parent->b_f[i].p_offset==deloff) break;
  neighbor_index=i-1;
  neighbor_offset=parent->b_f[i-1].p_offset;
  k_prime_index=i;
  k_prime=parent->b_f[i].key;
}

page*neighbor=load_page(neighbor_offset);
intmax=not_enough->is_leaf?LEAF_MAX:INTERNAL_MAX-1;
intwhy=neighbor->num_of_keys+not_enough->num_of_keys;
//printf("%d %d\n",why, max);
if (why<=max) {
  free(not_enough);
  free(parent);
  free(neighbor);
  coalesce_pages(deloff,neighbor_index,neighbor_offset,parent_offset,k_prime);
}
```

```
    else {
      free(not_enough);
      free(parent);
      free(neighbor);

    redistribute_pages(deloff,neighbor_index,neighbor_offset,parent_offset,k_prime,k
_prime_index);
    }

    return;
}
```

최소키 제약 조건은 if (not_enough->num_of_keys>=check)에서 비교를 통해 확인하고 있습니다. merge를 미루어 확실한 경우에만 merge하여 최소키 제약 조건을 만족할 수 있도록 int check의 크기를 조절하는 방식을 사용해보고자 하였습니다. 이번 테스트에서는 2/3 줄여서 merge를 지연시켜보았습니다.



자료를 50000개를 넣었을 때를 제외하면 최소키 제약 조건이 max의 1/2일 때가 속도가 더 빨랐습니다. 때문에 최소키 제약조건을 수정하여 merge를 지연시키는 것은 효과적인 방법이 아니라는 것을 알게 되었습니다.

이후 delay merge 방식을 2)의 임시 저장장치 방식으로 진행하는 것이 맞겠다는 생각을 가졌습니다. delete한 offset의 정보를 배열에 저장한 뒤, 일정 개수를 넘었을 때 각각의 page가 조건을 만족하는지 확인하였습니다. 여러 번 merge 해줘야했던 page의 경우에도 한 번의 merge로 줄일 수 있어서 효과를 얻을 수 있으리라고 생각했습니다.

```
intdb_delete(int64_tkey,intch) {


    if (rt->num_of_keys==0) {
      //printf("root is empty\n");
      return-1;
    }
    char*check=db_find(key);
    if (check==NULL) {
      free(check);
      //printf("There are no key to delete\n");
      return-1;
    }
    free(check);
```

```
    off_tdeloff=find_leaf(key);
    delete_entry(key,deloff,ch);
    return0;

}//fin
off_tchecker[10000];

intinit_delete_entry(intch) {
    for(inti=0; i<ch; i++) {
      page*not_enough=load_page(checker[i]);
      intcheck=not_enough->is_leaf?cut(LEAF_MAX) :cut(INTERNAL_MAX);
      if (not_enough->num_of_keys>=check){ //* dogyeom
        free(not_enough);
        //printf("just delete\n");
        return;
      } //최소키 조건

      intneighbor_index,k_prime_index;
      off_tneighbor_offset,parent_offset;
      int64_tk_prime;
      parent_offset=not_enough->parent_page_offset;
      page*parent=load_page(parent_offset);
      //이쪽을 고쳐야 delay-merge가 가능할듯
      if (parent->next_offset==checker[i]) {
        neighbor_index=-2;
        neighbor_offset=parent->b_f[0].p_offset;
        k_prime=parent->b_f[0].key;
        k_prime_index=0;
      }
      elseif(parent->b_f[0].p_offset==checker[i]) {
        neighbor_index=-1;
        neighbor_offset=parent->next_offset;
        k_prime_index=0;
        k_prime=parent->b_f[0].key;
      }
      else {
        inti;

        for (i=0; i<=parent->num_of_keys; i++)
          if (parent->b_f[i].p_offset==checker[i]) break;
```

```
      neighbor_index=i-1;
      neighbor_offset=parent->b_f[i-1].p_offset;
      k_prime_index=i;
      k_prime=parent->b_f[i].key;
    }

    page*neighbor=load_page(neighbor_offset);
    intmax=not_enough->is_leaf?LEAF_MAX:INTERNAL_MAX-1;
    intwhy=neighbor->num_of_keys+not_enough->num_of_keys;
    //printf("%d  %d\n",why,  max);
    if  (why<=max)  {
      free(not_enough);
      free(parent);
      free(neighbor);

    coalesce_pages(checker[i],neighbor_index,neighbor_offset,parent_offset,k_prime
);
    }
    else  {
      free(not_enough);
      free(parent);
      free(neighbor);

    redistribute_pages(checker[i],neighbor_index,neighbor_offset,parent_offset,k_pri
me,k_prime_index);
    }
  }


  return;
}



voiddelete_entry(int64_tkey,off_tdeloff,intch)  {

  remove_entry_from_page(key,deloff);

  if  (deloff==hp->rpo)  {
    adjust_root(deloff);
```

```
    return;
  }
  else {
    checker[ch] =deloff;
    return;
  }
}
```

void db_delete에 ch라는 현재 삭제가 몇 개 진행되었는지를 표시해주는 인자를 추가해 주었습니다. delete_entry에 ch를 전달해 주어 root일 때를 제외하고는 checker라는 배열에 값을 입력해주었습니다.

```
case'k':
        gettimeofday(&tv,NULL);
        scanf("%d",&k);
        doublestart2= (tv.tv_sec) *1000+ (tv.tv_usec) /1000;
        for(j=1; j<k; j++) {
          sprintf(buf,"%d",k-j);
          db_insert(j,buf);
          printf("%d Done!\n",j);
        }
        for(j=1; j<k; j++) {
          //sprintf(buf, "%d", k-j);
          if(ch>10) {
            init_delete(ch);
            ch=0;
          }
          db_delete(j,ch);
          printf("%d Done!\n",j);
        }
        gettimeofday(&tv,NULL);
        doubleend2= (tv.tv_sec) *1000+ (tv.tv_usec) /1000;
        printf("%d> Time: %f\n",++i, (end2-start2)/1000);//
        break;
```

이후 main문에서 delete가 진행될 때 마다 int ch에 값을 추가하여 10개가 삭제되었을 때 마다 해당 init_delete라는 checker 배열에 저장된 offset data를 기반하여 트리를 재구성하는 함수를 실행하였습니다. 현재의 코드는 삭제가 10번 진행되었을 때 배열을 재정리하도록 하였습니다.

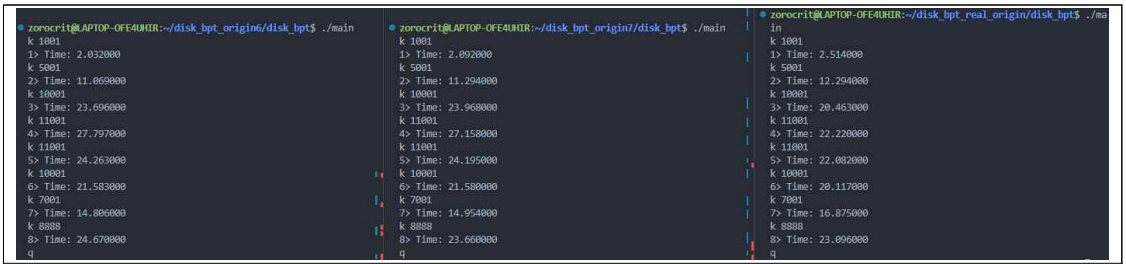
우측이 수정 전, 좌측이 수정 후의 실험 결과입니다. 500개, 10000개를 넣었을 때는 시간을 감축시키는 효과가 있었지만, 1000000개를 넣었을 때는 시간을 감축시키는 효과가 없었습니다. 원인을 생각해보자면 500개, 10000개 까지는 10개 단위로 delete를 진행했을 때 이점을 얻을 수 있었지만, 1000000의 경우에는 그 이점이 미미하여 오히려 memory에 부담을 주었던 것이라 생각했습니다.

때문에 100개 단위로 delete를 해주었습니다. 이 경우에도 이점을 갖지는 못하고 오히려 역효과를 가져왔습니다.

```
if (not_enough->num_of_keys>=check){ //* dogyeom
        free(not_enough);
        //printf("just delete\n");
        return;
    } //최소키 조건
```

문제를 확인해본 결과, if문에 return이 있어서 재조정이 안 되고 있었음을 확인하였습니다. 때문에 첫 테스트에서 시간 감축 효과가 있던 것은 merge를 delay하고 있었기 때문이 아닌, merge를 전혀 하지 않았기 때문이라는 결과를 얻었습니다.

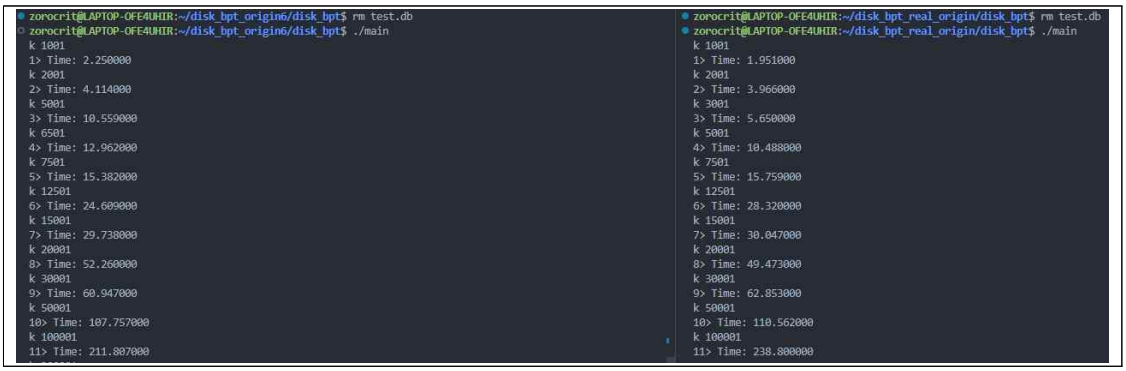
때문에 이 부분을 수정하여 다시 테스트를 진행해보았습니다. 10000개의 자료를 삽입하고 삭제했을 때는 1초 시간이 감소했지만, 30000개의 자료를 삽입하고 삭제했을 때는 delay merge로 얻을 수 있는 overhead 감소 효과가 확실히 있는지 확인하기 어려웠습니다. 100개 단위로 tree를 재조정한다면 tree 내 자료의 크기에 따라 disk IO를 감소시키는 부분도 있지만, 오히려 disk IO를 증가시키는 부분이 있을 수도 있겠다는 생각을 하게 되었습니다. 때문에 확실한 기준점을 찾기는 어렵더라도 tree의 크기에 따른 기준점의 변화가 반드시 필요하다는 생각을 갖게 되었습니다.



왼쪽이 10개를 기준으로 tree 재조정, 중앙이 100개를 기준으로 tree를 재조정, 오른쪽은 기존 그대로 실행한 결과입니다. 1000개의 자료를 삽입하고 삭제했을 때는 10개의 delete를 기준으로 tree를 재조정했을 때 가장 빨랐습니다. 5000개의 자료를 사용했을 때, 그리고 7000개의 자료를 사용했을 때에도 10개의 delete를 기준으로 tree를 재조정했을 때가 가장 빨랐습니다. 하지만 8000개가 넘어갈 때부터 기존 그대로의 b+tree를 사용했을 때가 가장 빨랐습니다.

```
case'k':
        gettimeofday(&tv,NULL);
        scanf("%d",&k);
        doublestart2= (tv.tv_sec) *1000+ (tv.tv_usec) /1000;
        for(j=1; j<k; j++) {
          sprintf(buf,"%d",k-j);
          db_insert(j,buf);
          printf("%d Done!\n",j);
        }
        for(j=1; j<k; j++) {
          //sprintf(buf, "%d", k-j);
          if(num>100) {
            if(ch>num/100) {
              init_delete(ch);
              ch=0;
            }
            num=db_delete(j,ch);
            printf("%d Done!\n",j);
          }
          else {
            db_delete2(j);
            printf("%d Done!\n",j);

          }
        }
        gettimeofday(&tv,NULL);
        doubleend2= (tv.tv_sec) *1000+ (tv.tv_usec) /1000;
        printf("%d> Time: %f\n",++i, (end2-start2)/1000);//
        break;
```

따라서 전체 자료가 100개 이전에는 기존의 delete 방식을 사용하고, 전체 자료가 100개를 넘어갈 때부터 삭제된 자료의 수가 (전체 자료)/100을 넘을 때 tree를 재조정하는 방식으로 b+tree를 수정해보았습니다.



초반에는 자료의 개수를 불러오는 과정과 조건문이 추가되어 시간이 증가하지만, 이후 자료의 크기에 따른 delay merge를 해주는 과정에서 점차 시간적인 장점을 얻어가는 모습을 볼 수 있습니다.

다만 지정해둔 배열의 크기가 10000개 제한이 있으므로 100만개 이상의 자료의 연산에서는 이러한 방식으로 진행 할 수 없습니다.

```
case'k':
        gettimeofday(&tv,NULL);
        scanf("%d",&k);
        doublestart2= (tv.tv_sec) *1000+ (tv.tv_usec) /1000;
        for(j=1; j<k; j++) {
          sprintf(buf,"%d",k-j);
          db_insert(j,buf);
          //printf("%d Done!\n",j);
        }
        for(j=1; j<k; j++) {
          //sprintf(buf, "%d", k-j);
          if(num>100000) {
            if(ch>num/800) {
              init_delete(ch);
              ch=0;
            }
            num=db_delete(j,ch);
            ch++;
          }
          elseif(num>100) {
            if(ch>num/100) {
              init_delete(ch);
              ch=0;
            }
            num=db_delete(j,ch);
            ch++;
            //printf("%d Done!\n",j);
          }
          else {
            db_delete2(j);
            //printf("%d Done!\n",j);

          }
        }
        gettimeofday(&tv,NULL);
```

```
        doubleend2= (tv.tv_sec) *1000+ (tv.tv_usec) /1000;
        printf("%d> Time: %f\n",++i, (end2-start2)/1000);//
        break;
```

때문에 자료의 크기에 따라 delay merge를 해주는 빈도를 수정해보았습니다.

좌측이 2가지의 delay merge를 활용한 테스트이고, 우측이 기존 b+tree를 실행한 내용입니다. 자료 300000개를 삽입하고 삭제했을 때, 2가지 delay merge를 사용한 테스트가 30초가량 일찍 끝났음을 확인 할 수 있습니다.

/home/zorocrit/disk_bpt_final/disk_bpt/src/main.c

5. Design

-각 Disk를 순회할 때 마다 overhead가 많이 발생함을 배웠습니다. 때문에 선형 탐색이 이진 탐색에 비해 Disk IO를 적게 발생시켜 오히려 이진 탐색 알고리즘이 선형 탐색 알고리즘에 비해 더 많은 시간을 소요함을 알 수 있었습니다. 하지만, Tree가 갖고 있는 자료가 50만개를 넘어서면서부터 이진 탐색이 발생하는 Disk io의 부담을 지더라도 선형 탐색에 비해 더욱 효과적임을 발견하였습니다. 따라서 자료의 크기에 따라 선형 탐색과 이진 탐색을 선택적으로 진행할 수 있도록 db_find 함수를 수정하였습니다.

-매 delete마다 트리의 최솟값 규정을 지키며 merge를 진행해주다보니 overhead가 많이 발생됨을 알게 되었습니다. 따라서 delete 할 때 마다 최솟값 규정을 만족하는지 측정하지 않고, 일정한 제약 조건을 부여하여 조건을 만족할 때 마다 delete가 최솟값 규정을 만족하는지 확인하였습니다. 이러한 방식을 사용하면 merge가 발생하는 횟수를 줄일 수 있어 overhead를 감소시킬 수 있습니다.

6. Implement

main.c

```
case'd':
        scanf("%ld",&input);
        if(num<=100) {
          //printf("%d dele!\n",j);
          db_delete2(input);
          break;
        }
        elseif(num>100000) {
          if(ch>num/800) {
            init_delete(ch);
            ch=0;
          }
          num=db_delete(input,ch);
```

```
        ch++;
        break;
      }
      else {
        if(ch>num/100) {
          init_delete(ch);
          ch=0;
        }
        num=db_delete(input,ch);
        ch++;
        //printf("%d Done!\n",j);
        break;
      }
```

int main() 안에 지역 변수로
int ch = 0;
int num = -1;
을 선언해주었습니다.
case ‘d’의 경우 tree의 전체 크기에 따라 다른 방식으로 작동합니다. tree의 전체 자료수가 100개를 넘지 않았을 때는 기존의 delete 방식인 db_delete2를 사용합니다. tree의 전체 자료 수가 100개를 넘어가면서부터 num에 tree에 있는 자료의 수를 저장하고 ch에 이후부터 작동한 delete의 개수를 측정합니다.

num <= 100: db_delete2를 이용하여 기존 삭제 방식을 따른다.
num > 100, num <= 100000: 삭제된 자료의 수(ch)가 num/100을 넘었을 때 init_delete 함수를 호출하여 삭제된 자료를 가진 page가 최솟값 조건을 만족할 수 있도록 해줍니다.
num > 100000: 삭제된 자료의 수(ch)가 num/800을 넘었을 때 init_delete 함수를 호출하여 삭제된 자료를 가진 page가 최솟값 조건을 만족할 수 있도록 해줍니다.

bpt.h

```
intdb_delete(int64_tkey,intch);
voiddelete_entry(int64_tkey,off_tdeloff,intch);
intinit_delete(intch);
intdb_delete2(int64_tkey);
voiddelete_entry2(int64_tkey,off_tdeloff);
```

db_delete는 tree의 전체 자료수가 100개를 넘을 때부터 사용된다. 자료가 현재 tree 내에 있는지 확인하고 delete_entry를 호출한다.
delete_entry는 tree내에서 명령받은 자료를 삭제하고, buffer에 삭제된 자료를 저장한다.
init_delete는 tree의 전체 자료수가 100을 넘을 때부터 사용된다. ch와 num이 특정 조건을 만족할 때, 전체 tree 내 최솟값 조건을 만족하지 않는 page에 대해서 merge를 진행한다.

bpt.c

```
char*db_find(int64_tkey) {
  char*value= (char*)malloc(sizeof(char) *120);
  inti=-1;
  off_tfin=find_leaf(key);
  if (fin==0) {
   returnNULL;
  }
  page*p=load_page(fin);
  if(p->num_of_keys>=500000) {
   //* dogyeom
   intlow=0;
   inthigh=p->num_of_keys-1;
   intmid;
   while(low<=high) {
    mid= (low+high) /2;
    if(p->records[mid].key==key) {
     i=mid;
     break;
    }
    elseif(p->records[mid].key>key) {
     high=mid-1;
    }
    else {
     low=mid+1;
    }
   }
   if (i==-1) {
    free(p);
    returnNULL;
   }
   else {
    strcpy(value,p->records[i].value);
    free(p);
    returnvalue;
   }
  }
  else {
   for (; i<p->num_of_keys; i++) {
    if (p->records[i].key==key) break;
   }
```

```
    if (i==p->num_of_keys) {
      free(p);
      returnNULL;
    }
    else {
      strcpy(value,p->records[i].value);
      free(p);
      returnvalue;
    }
  }
}
```

db_find는 tree의 자료 수에 따라 선형 탐색과 이진 탐색을 선택적으로 진행 할 수 있도록 설계되어있습니다. 때문에 tree의 자료 수가 500000 이상일 때는 이진 탐색을 진행하며, 그 이하일 때는 선형 탐색을 진행합니다.

```
intdb_delete(int64_tkey,intch) {

  if (rt->num_of_keys==0) {
    //printf("root is empty\n");
    return-1;
  }
  char*check=db_find(key);
  if (check==NULL) {
    free(check);
    //printf("There are no key to delete\n");
    return-1;
  }
  free(check);
  off_tdeloff=find_leaf(key);
  delete_entry(key,deloff,ch);
  returnrt->num_of_keys;

}//fin
```

```
off_tchecker[10000];

intinit_delete(intch) {
  for(inti=0; i<ch; i++) {
    page*not_enough=load_page(checker[i]);
    intcheck=not_enough->is_leaf?cut(LEAF_MAX) :cut(INTERNAL_MAX);
    if (not_enough->num_of_keys>=check){ //* dogyeom
      free(not_enough);
      //printf("just delete\n");
```

```
    //return;
  } //최소키 조건
  else {
    intneighbor_index,k_prime_index;
    off_tneighbor_offset,parent_offset;
    int64_tk_prime;
    parent_offset=not_enough->parent_page_offset;
    page*parent=load_page(parent_offset);
    //이쪽을 고쳐야 delay-merge가 가능할듯
    if (parent->next_offset==checker[i]) {
      neighbor_index=-2;
      neighbor_offset=parent->b_f[0].p_offset;
      k_prime=parent->b_f[0].key;
      k_prime_index=0;
    }
    elseif(parent->b_f[0].p_offset==checker[i]) {
      neighbor_index=-1;
      neighbor_offset=parent->next_offset;
      k_prime_index=0;
      k_prime=parent->b_f[0].key;
    }
    else {
      inti;

      for (i=0; i<=parent->num_of_keys; i++)
        if (parent->b_f[i].p_offset==checker[i]) break;
      neighbor_index=i-1;
      neighbor_offset=parent->b_f[i-1].p_offset;
      k_prime_index=i;
      k_prime=parent->b_f[i].key;
    }

    page*neighbor=load_page(neighbor_offset);
    intmax=not_enough->is_leaf?LEAF_MAX:INTERNAL_MAX-1;
    intwhy=neighbor->num_of_keys+not_enough->num_of_keys;
    //printf("%d %d\n",why, max);
    if (why<=max) {
      free(not_enough);
      free(parent);
      free(neighbor);
```

```
      coalesce_pages(checker[i],neighbor_index,neighbor_offset,parent_offset,k_prim
e);
      }
      else {
        free(not_enough);
        free(parent);
        free(neighbor);

      redistribute_pages(checker[i],neighbor_index,neighbor_offset,parent_offset,k_p
rime,k_prime_index);
      }
    }
  }


  return;
}
```

```
voiddelete_entry(int64_tkey,off_tdeloff,intch) {

  remove_entry_from_page(key,deloff);

  if (deloff==hp->rpo) {
    adjust_root(deloff);
    return;
  }
  else {
    checker[ch] =deloff;
    return;
  }
}
```

case ‘d’에서 수행되는 연산입니다.
num(tree의 자료 수)이 100 초과일 때부터 사용됩니다.
db_delete: root가 null인지 확인하고, key가 tree 내 존재하는지 안하는지를 확인합니다. 이후 delete_entry를 호출합니다.
delete_entry: remove_entry_from_page를 호출합니다. head pointer라면 adjust_root를 호출합니다. 다른 경우에는 checker 배열에 삭제한 off_t를 저장합니다.
init_delete: checker 배열에 저장된 off_t값을 ch 값 만큼 순회하며 tree를 최솟값 조건에 맞게 merge 해줍니다. merge를 delay하여 한 번에 계산하는 덕분에 발생하는 merge 횟수가 감소합니다.

7. Result

```
zorocrit@LAPTOP-OFE4UHIR:~/disk_bpt_final/disk_bpt$ rm test.db
zorocrit@LAPTOP-OFE4UHIR:~/disk_bpt_final/disk_bpt$ ./main
k 800001
```

test.db를 제거해준 후 main을 실행시켜줍니다.

```
zorocrit@LAPTOP-OFE4UHIR:~/disk_bpt_final/disk_bpt$ ./main
k 800001
1> Time: 1967.505000
k 10001
2> Time: 24.185000
k 300001
3> Time: 760.164000
```

```
zorocrit@LAPTOP-OFE4UHIR:~/disk_bpt_real_origin/disk_bpt$ ./main
k 800001
1> Time: 1970.197000
k 10001
2> Time: 25.740000
k 300001
3> Time: 760.194000
```

```
zorocrit@LAPTOP-OFE4UHIR:~/disk_bpt_final/disk_bpt$ rm test.db
zorocrit@LAPTOP-OFE4UHIR:~/disk_bpt_final/disk_bpt$ ./main
f 1
Not Exists
f 2
Not Exists
f 3
Not Exists
i 1 2
i 2 3
i 3 4
i 4 5
f 1
Key: 1, Value: 2
f 2
Key: 2, Value: 3
f 3
Key: 3, Value: 4
f 4
Key: 4, Value: 5
d 1
d 2
d 3
f 1
Not Exists
f 2
Not Exists
f 3
Not Exists
f 4
Key: 4, Value: 5
q
zorocrit@LAPTOP-OFE4UHIR:~/disk_bpt_final/disk_bpt$
```