Journal of Korea Game Society 2020 Oct; 20(5): 53-64
http://dx.doi.org/10.7583/JKGS.2020.20.5.53

컴퓨터 그래픽스(공학)

HD-Tree: 고성능 Lock-Free NNS KD-Tree

이상기[*], 정내훈[**]
한국산업기술대학교 디지털엔터테인먼트학과[*], 한국산업기술대학교 게임공학부[**]
{lunta94, nhjung}@kpu.ac.kr

HD-Tree: High performance Lock-Free Nearest Neighbor Search KD-Tree

Sang-gi Lee[*], NaiHoon Jung[**]
Dept. of Digital Entertainment, Korea Polytechnic Univ.[*], Dept. of Game & Multimedia Engineering, Korea Polytechnic Univ.[**]

요 약

KD-Tree에서 NNS의 구현은 다차원 데이터를 다루는 응용 프로그램에서 필수적이다. 본 논문에서는 자료구조의 동시 수정, 검색이 일어나는 멀티스레드 상황에서 NNS를 지원하는 고성능 Lock-Free KD-Tree인 HD-Tree를 제안한다. HD-Tree는 동기화에 사용되는 노드 수를 최소화하고, 사용하는 원자 연산자의 수를 감소시켜 성능을 개선하였다. 실험 결과 HD-Tree는 8코어 16스레드의 멀티코어 시스템에서 기존의 NNS보다 성능이 최대 95% 향상되었고, 삽입/삭제연산은 코어보다 스레드가 많은 상황에서 기존 알고리즘보다 최대 15%향상된 성능을 보여준다.

ABSTRACT

Supporting NNS method in KD-Tree algorithm is essential in multidimensional data applications. In this paper, we propose HD-Tree, a high-performance Lock-Free KD-Tree that supports NNS in situations where reads and writes occurs concurrently. HD-Tree reduced the number of synchronization nodes used in NNS and requires less atomic operations during Lock-Free method execution. Comparing with existing algorithms, in a multi-core system with 8 core 16 thread, HD-Tree's performance has improved up to 95% on NNS and 15% on modifying in oversubscription situation.

Keywords : KD-Tree(KD트리), Lock-Free(무잠금), Data structure(자료구조), Nearest Neighbor Search(가장 가까운 이웃 검색)

Received: Sep. 20. 2020 Revised: Oct. 08. 2020
Accepted: Oct. 14. 2020
Corresponding Author: Sang-gi Lee(Korea Polytechnic Univ.)
E-mail: lunta94@kpu.ac.kr

ISSN: 1598-4540 / eISSN: 2287-8211

Ⓒ The Korea Game Society. All rights reserved. This is an open-access article distributed under the terms of the Creative Commons Attribution Non-Commercial License (http://creativecommons.otg/licenses/by-nc/3.0), which permits unrestricted non-commercial use, distribution, and reproduction in any medium, provided the original work is properly cited.

1. 서 론

최근 게이밍 PC에 사용되는 CPU의 코어 수가 급격히 늘어나고 있다. 스팀의 하드웨어 통계[4]를 보면 싱글 코어에서부터 쿼드 코어까지 4코어 이하의 CPU 제품들의 비중이 꾸준히 감소세를 보이고 있고, 6코어 이상의 CPU 제품 비중이 꾸준히 늘어나고 있는 것을 확인할 수 있다. 이와 같이 멀티코어 시스템이 급속도로 보급됨에 따라, 이를 충분히 활용할 수 있는 효율적인 병행 알고리즘의 필요성이 증가하고 있으며, 큐와 리스트 같은 자주 사용되는 일반적인 자료구조들은 이미 굉장히 많은 수의 효율적인 병행 알고리즘이 제안되었다[2].

하지만 가장 일반적으로 사용되는 다차원 자료구조인 KD-Tree의 병행 알고리즘은 CPU의 멀티코어 시스템을 활용하기 보다는 주로 그래픽스 분야에서 SIMD(Single-Instruction-Multiple-Data)나 GPU를 사용한 하드웨어 가속을 활용하는 방향으로 발전해왔다[1].

KD-Tree는 게임, 기계학습, 이미지검색 등 컴퓨터 그래픽스를 활용한 분야에서 주로 사용된다. 이러한 분야에서 다차원 데이터를 활용하기 위해 일반적으로 NNS(Nearest Neighbor Search)를 사용하기 때문에 멀티코어 시스템에서 이를 제대로 활용하기 위해서는 검색, 삽입, 삭제 작업이 병행하게 일어나는 상황에서 NNS를 지원해야한다.

본 논문에서는 멀티코어 시스템을 충분히 활용할 수 있는 고성능 Lock-Free NNS KD-Tree인 HD-Tree를 제안한다. HD-Tree는 기존에 제안된 JAVA로 구현된 병행 KD-Tree[1]를 C++로 변환한 것을 기반으로 하였다. C++에서의 성능 개선을 위해 새로운 메모리 재사용 구조를 추가하고, 개선된 동기화 방법을 사용하여 기존 알고리즘이 가지고 있는 성능상의 문제를 해결하였다.

본 논문은 2장에서 관련연구에 대해 설명하고, 3장에서 HD-Tree 알고리즘을 구현한다. 4장에서는 구현된 알고리즘을 실험하고 평가한다. 마지막으로 5장에서 결론과 향후 연구방향을 제시한다.

2. 관련연구

이번 장에서는 HD-Tree구현의 기반이 되는 Lock-Free 알고리즘과 기술자(Descriptor) 기반 동기화 방법에 대해 설명한 후, HD-Tree의 비교대상인 Lock-Free NNS KD-Tree의 구현 방식과 문제점에 대해 설명하도록 한다.

2.1 Lock-Free 알고리즘

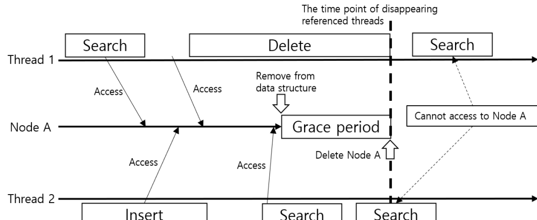
병행 알고리즘은 공유메모리의 메모리 일관성을 유지하기 위해 주로 잠금을 사용하거나 CAS (Compare And Swap)와 같은 원자 연산자를 이용하여 구현된다. 이 중에서 잠금을 사용하여 구현된 병행 알고리즘은 잠금에 의한 블로킹으로 인해 여러 취약점[5]을 가지고 있고, 이러한 이유 때문에 원자 연산자만을 이용해서 블로킹이 없도록 구현되는 논블로킹 알고리즘이 선호되고 있다. 여러 논블로킹 알고리즘 중에서 Lock-Free방법[7]이 가장 많이 사용되고 있고, 본 논문의 HD-Tree또한 Lock-Free로 구현된다.

Lock-Free 알고리즘은 주로 포인터의 최하위 비트(Least Significant Bit, LSB)에 기술자들을 끼워 넣어 한 번의 CAS연산으로 포인터 값과 노드의 상태를 동시에 원자적으로 변경하는 기술자 기반 동기화 방법[8]을 많이 사용한다.

이 방법은 32비트 CPU의 경우 어플리케이션이 사용하는 포인터에서 최하위 비트 중 일부는 대부분의 경우 사용되지 않는다는 것을 이용하고 있다. 32비트 아키텍처에서 워드는 4바이트이고, 워드 크기로 정렬된 메모리의 주소 값은 항상 4의 배수이므로 비트로 나타냈을 때 언제나 00으로 끝나기 때문에 마지막 2비트를 다른 용도로 사용할 수 있다. 같은 이유로 64비트 아키텍처에서는 워드 크기가 8바이트이므로 3비트를 사용할 수 있다.

기술자 기반 동기화 방법을 사용할 경우 노드를 제거할 때 다른 스레드가 제거될 노드에 접근하여 발생하는 제거 타이밍 문제와 ABA문제[8]가 존재

한다. 때문에 Lock-Free 자료구조를 구현할 경우 hazard pointer[9], EBR(Epoch Based memory Reclamation)[3]과 같은 메모리를 관리를 위한 Lock-Free 알고리즘이 추가적으로 필요하다.

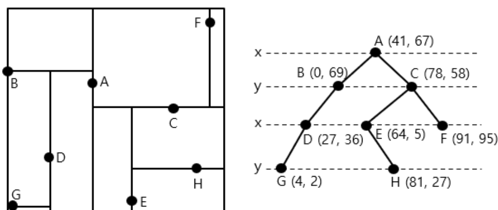


[Fig. 1] Epoch Based memory Reclamation

본 논문에서는 노드를 물리적으로 제거할 때 모든 스레드가 참조를 끝마칠 때 까지 유예기간을 두어 노드를 안전하게 제거할 수 있도록 도와주는 EBR알고리즘을 사용한다. [Fig. 1]에서는 EBR알고리즘에서 유예기간을 어떤 시점에 적용하는지를 보여주고 있다.

2.2 Lock-Free NNS KD-Tree

J. L. Bentley가 최초로 제안한 KD-Tree[6]는 BST(Binary Search Tree)의 일종이며 트리의 모든 노드는 k차원(k-dimension, KD) 공간에 존재한다. 예를 들어, k=2일 경우 모든 노드는 2차원 점을 키 값으로 가진다. KD-Tree는 주어진 다차원 데이터를 이용해서 공간을 한 번에 하나의 축을 따라 분할하고, 트리의 계층마다 분할하는 기준 축을 변경한다. KD-Tree는 이러한 과정을 통해 분할 계층구조를 구성하여 다차원 데이터를 분류한다.

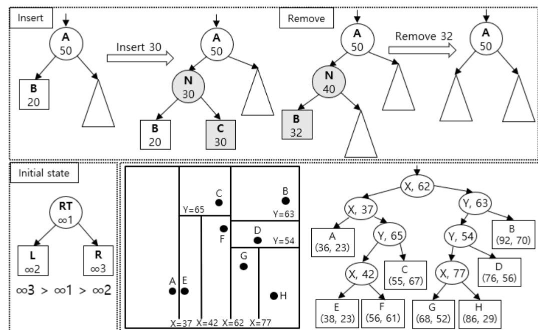


[Fig. 2] KD-Tree

KD-Tree는 일반적으로 각 노드가 최대 하나의 다차원 데이터를 저장하며, 저장된 데이터를 키 값으로 사용한다. [Fig. 2]를 보면 트리의 계층마다 정해지는 기준 축에 해당하는 키 값으로 기준 축에 직교하는 초평면을 만들어 공간을 분할하는 것을 확인할 수 있다.

하지만 모든 노드에 데이터를 저장하는 일반적인 BST는 노드를 제거할 때 비용이 많이 들며 Lock-Free구현을 할 경우 훨씬 더 큰 비용을 지불해야한다[1]. 때문에, BST의 Lock-Free구현을 할 때 대부분의 경우 [10,11]에서 설명하는 것처럼 데이터가 모두 리프노드에 저장되고 내부노드들은 검색 작업에서 키 값을 분류하는 역할만 하는 외부트리(External tree) 방식으로 노드를 관리한다.

[Fig. 3]에서 볼 수 있듯이 외부트리의 모든 리프노드가 채워져 있기 때문에 노드를 삽입할 때 새로운 노드를 삽입할 위치에 반드시 기존 노드가 존재한다. 그래서 한 번 삽입할 때 새 데이터를 저장할 리프노드 뿐만 아니라 기존 노드와 새 노드를 키 값에 따라 분류할 내부노드를 함께 만들어 트리에 연결한다. 노드를 제거할 때도 마찬가지로 제거할 노드에 연결된 내부노드를 함께 제거하고 형제노드를 내부노드의 부모노드에 연결하여 후처리를 한다.



[Fig. 3] External tree

이와 같이, 외부트리 방식을 사용하면 일반적인 방식의 BST가 노드를 제거할 때 목표 노드의 자식 노드의 개수에 따라서 여러 가지 경우의 수를 고려해야 하는 것과 다르게, 위에서 설명한 한 가지의 경우의 수만 고려하면 되기 때문에 비교적 쉽게 Lock-Free구현을 할 수 있다.

NNS는 자료구조에서 주어진 다차원 키와 유클리드 거리가 가장 가까운 점을 찾는 알고리즘이다. KD-Tree에서의 NNS는 컴퓨터 그래픽스 분야의 광선추적 알고리즘, 머신러닝에서의 학습 데이터 분류 등에 사용된다.

KD-Tree에서의 NNS알고리즘은 다음과 같다.

1) 점 P의 키 값으로 검색했을 때 처음 도달한 리프노드를 NN(Nearest Neighbor)으로 갱신하고 P와 현재노드(리프노드) 사이의 거리 r을 구한다.

2) 현재노드가 루트와 같으면 찾은 NN을 반환한다.

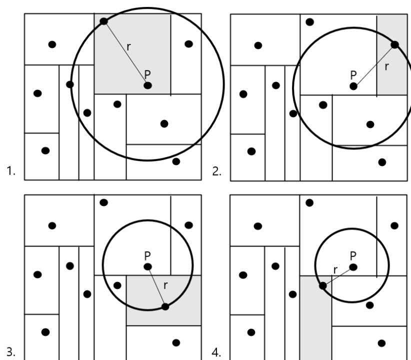
3) 부모노드의 키 값과 P 사이의 거리가 r보다 클 경우, 현재노드를 부모노드로 갱신하고 2)부터 다시 시도한다.

4) 부모노드 키 값과 P 사이의 거리가 r보다 작을 경우, 부모노드의 형제노드에서부터 P를 키 값으로 검색하여 도달한 리프노드와 거리 r을 구한다.

5) 새로 구한 r값이 이전 값보다 작을 경우 리프노드를 NN으로 갱신하고 2)부터 다시 시도한다.

6) 새로 구한 r값이 이전 값보다 클 경우 현재노드를 부모노드로 갱신하고 2)부터 다시 시도한다.

[Fig. 4]는 NNS알고리즘이 진행되는 방식을 보여주고 있다.



[Fig. 4] Nearest Neighbor Search

최근 B. Chatterjee등 3인[1]은 최초로 병행 수정, 검색이 일어나는 상황에서 NNS를 지원하는 Lock-Free NNS KD-Tree를 JAVA로 구현하였다. 본 논문은 이를 기반으로 하고 있으며, 이후 이 알고리즘을 기존 알고리즘이라고 칭한다.

기존 알고리즘은 기술자 기반 동기화 방법을 사용하며, 삭제에서 세 가지의 기술자를 사용하여 알고리즘을 구현하였다. 또한, NNS의 Lock-Free 알고리즘을 구현하기 위해 NNC(Nearest Neighbor Collector) 리스트를 사용한다.

기존 알고리즘을 메모리 관리가 자동으로 되지 않는 언어인 C++로 구현할 경우 NNS에서 성능상의 문제가 발생한다. NNS를 실행한 후, NNS를 끝마치기 전에 사용한 NNC노드를 비활성화하고 NNC리스트에서 제거를 시도하며 노드를 제거하지 못하더라도 훗날을 기약하고 NNS를 끝마치게 되는데, 이후 제거가 되지 않은 노드에 대한 후처리를 하는 부분이 없기 때문에 NNS를 호출하면 호출할수록 NNC리스트에 남아있는 노드가 점점 증가하게 되어 시간이 지날수록 검색 성능이 떨어지는 문제가 발생한다.

HD-Tree는 이러한 성능상의 문제를 해결하고 구조를 개선하여 높은 성능을 얻었다.

추가적으로, KD-Tree는 BST를 기반으로 하고 있기 때문에 효율적인 BST 알고리즘이 있으면 키 값을 비교할 때 다차원 데이터를 고려하는 루틴을 추가하는 것으로 BST 알고리즘을 KD-Tree로 변경할 수 있다. A. Natarajan등 3인[2]은 효율적인 Lock-Free BST구현을 위한 새로운 동기화 방법을 제안하였고, HD-Tree는 이를 사용하여 기존 알고리즘을 개선하였다.

3. HD-Tree의 구현

3.1 KD-Tree의 Lock-Free구현

HD-Tree에서의 검색은 기본적으로 일반적인 KD-Tree와 같이 주어진 다차원 키를 사용해 트리를 탐색한다.

검색 알고리즘은 다음과 같다.

1) Ancestor를 루트, Successor와 Parent를 루트의 왼쪽자식, Current를 Parent의 왼쪽자식으로

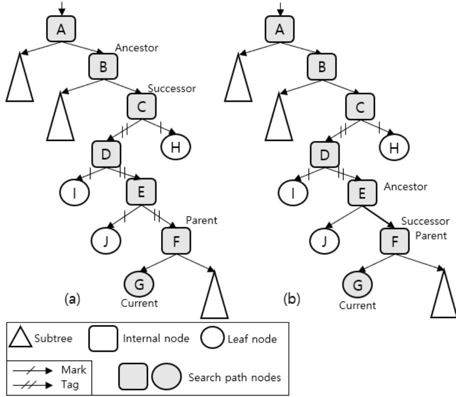
초기화한다.

2) Current가 리프노드이면 검색을 종료한다.

3) Parent의 LSB에 기술자가 삽입되어있지 않다면 Ancestor를 Parent로, Successor를 Current로 갱신한다.

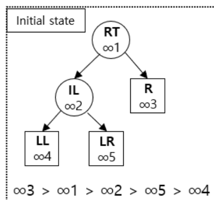
4) Parent를 Current로, Current를 Current의 주어진 키 방향 자식노드로 갱신하고 2)부터 다시 시도한다.

HD-Tree에서 노드의 LSB에 기술자가 삽입되어 있다면 해당 노드는 자료구조에서 제거해야하거나 재조정이 필요하여 해당 노드가 유효하지 않은 것을 의미한다.



[Fig. 5] Search in HD-Tree

HD-Tree의 검색이 기존 알고리즘과 다른 점은 한 번의 검색으로 네 가지의 노드를 찾는다는 점이다. [Fig. 5]는 HD-Tree에서 검색을 통해 얻은 노드들의 위치를 나타내고 있다. (a)에서는 검색도중 유효하지 않은 노드들이 연속적으로 나타날 경우 Ancestor와 Parent의 사이가 벌어지는 모습을 보이고 있고, (b)에서는 유효하지 않은 노드들이 연속적으로 나타나더라도 Parent의 부모노드가 유효할 경우의 검색결과를 나타내고 있다.



[Fig. 6] Initial state of HD-Tree

이러한 검색의 특수성 때문에 HD-Tree는 [Fig. 6]과 같이 초기 형태를 구성할 때 일반적인 외부트리와 다르게 루트의 왼쪽 자식에 더미 내부노드를 두어 검색을 최적화한다.

HD-Tree의 삽입에서 새로 생성되는 내부노드의 기준 축은 삽입 위치에 존재하는 노드와 새로 생성되는 노드의 키 값을 각각의 축을 기준으로 거리를 계산하여 가장 거리가 짧은 축을 기준 축으로 하고 키 값은 기준 축에서 두 노드의 키 값의 중간 값으로 한다.

삽입 알고리즘은 다음과 같다.

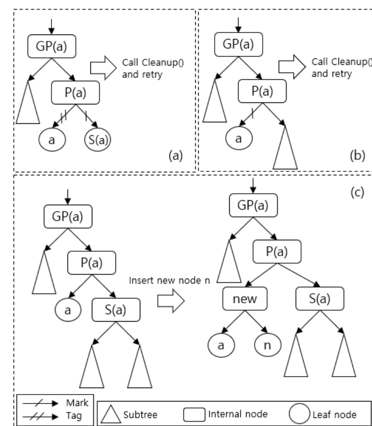
1) 검색을 통해 삽입할 위치의 노드를 찾는다.

2) 찾은 노드의 키 값이 주어진 키 값과 같으면 삽입에 실패하고 반환한다.

3) 찾은 노드의 LSB에 기술자가 삽입되어있지 않다면 CAS를 사용하여 새로운 내부노드와 리프노드를 생성하여 이전에 설명한 내부트리 방식으로 부모노드에 연결한다.

4) 3)에서 CAS가 실패했을 경우 1)부터 다시 시도한다.

5) 찾은 노드의 LSB에 기술자가 삽입되어있다면 후처리 함수를 호출한 후 1)부터 다시 시도한다.



[Fig. 7] Insert in HD-Tree

[Fig. 7]의 (a), (b)는 삽입 알고리즘에서 검색을 통해 얻은 노드가 유효하지 않을 경우 후처리를

하는 모습을 나타내고, (c)는 정상적으로 삽입이 진행되는 경우를 보여준다.

HD-Tree의 삭제는 기존 방법과는 다르게 제거할 노드를 표시하는 Mark와 재조정해야 하는 형제노드를 표시하는 Tag로 두 가지의 기술자를 사용한다.

삭제 알고리즘은 다음과 같다.

1) 검색을 통해 제거할 노드를 찾는다.

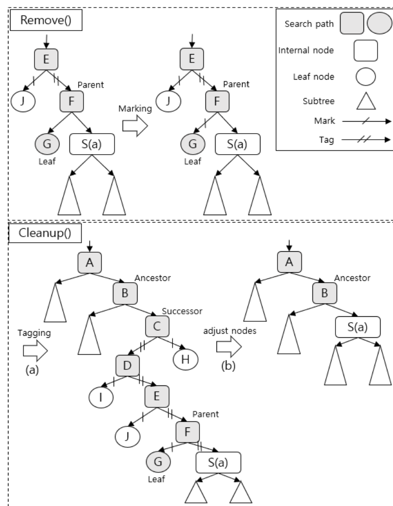
2) 찾은 노드의 키 값이 주어진 키 값과 다르면 제거를 종료한다.

3) 찾은 노드의 LSB에 기술자가 삽입되어있다면 후처리를 시도한 후 1)부터 다시 시도한다.

4) 찾은 노드의 LSB에 기술자가 삽입되어있지 않다면 Mark를 CAS를 통해 삽입한다.

5) 4)에서 CAS가 실패할 경우 1)부터 다시 시도한다.

6) 4)에서 CAS가 성공할 경우 후처리를 시도하고, 후처리가 성공할 경우 제거를 종료하고 실패할 경우 1)부터 다시 시도한다.



[Fig. 8] Remove in HD-Tree

후처리 단계에는 기술자가 삽입된 노드의 형제노드 LSB에 Tag가 없다면 삽입한 후 검색을 통해 찾아놓은 조상노드의 자식노드 위치에 형제노드를 연결하여 트리에서 연결되어있는 제거할 노드들을 한 번에 물리적으로 제거한다. [Fig. 8]은 HD-Tree의 제거와 후처리를 하는 모습을 나타낸다.

지금까지 설명한 HD-Tree의 수도코드는 [Fig. 9]에서 볼 수 있다. Node에서 d는 기준 축, c는 내부노드의 키 값을 의미하고 k는 노드에 저장된 다차원 키를 나타낸다. lt, rt, pr은 각각 노드의 왼쪽, 오른쪽, 부모노드의 링크를 의미한다. search_field는 검색에 사용된다. ac는 조상노드, sc는 조상노드의 자식노드이며 cr은 리프노드, pr는 리프노드의 부모노드이다. 변수 이름에 dir가 붙은 bool 변수는 자식노드가 부모노드의 어떤 방향에 있는지를 나타내며 true는 왼쪽, false는 오른쪽방향을 의미한다.

1-5라인은 HD-Tree를 [Fig. 6]처럼 초기화하는 과정을 수행하며 Search()의 6-11라인은 이렇게 초기화된 HD-Tree에서 검색의 초기 값을 설정한 후 12-19라인의 루프를 돌면서 주어진 키 값에 따른 리프노드를 찾는 것과 동시에 이전 루프의 부모노드의 LSB를 지속적으로 확인하며 조상노드와 그 자식노드를 갱신한다.

Contains()에서는 23라인에서 Search()를 통해 얻은 리프노드의 LSB를 확인하여 찾은 노드가 유효한지 확인하고 찾는 키 값과 같은 키를 가지고 있다면 검색에 성공하고 Sync()를 호출한 뒤 true를 반환한다. 이때 Contains(), Insert(), Remove() 함수의 처음과 끝에 있는 start_op()와 end_op()는 EBR알고리즘에서 스레드가 자료구조에 접근하고 있는지 여부를 판단할 때 사용되며 Contains(), Insert()함수에서 검색, 삽입 작업을 성공했을 때 호출하는 Sync()는 NNS알고리즘을 보조하기 위해 사용된다.

Insert()에서 33-47라인은 Search()를 통해 얻은 위치에 삽입할 새로운 노드들을 만드는 과정을 수행한다. 37라인의 GetMinDiffDimension()은 위에서 설명했던 삽입하는 내부노드의 기준 축을 계산하여 반환한다. 48라인에서는 자료구조에 새로운 노드를 연결하는 것을 CAS를 통해 시도하며 실패할 경우 리프노드의 LSB를 확인하여 기술자가 존

```
class Node { int d; double c; double k[]; Node* lt, rt, pr; }
class Neighbor { Node* p; double d; }
class NNCollector { double tgt[]; Neighbor* cn, rn; }
struct search_field { Node* ac; sc; pr; cr; bool sc_dir; cr_dir; }
struct seek_field { Node* pr; cr; bool cr_dir; double hi[]; double lo[]; }

class HD-Tree {
  bool Contains(double k[DIMENSION]);
  bool Insert(double k[DIMENSION]);        Member:
  bool Remove(double k[DIMENSION]);          Node* root;
  double[] NNS(double k[]);                  NNCollector* nncs[MAX_THREAD];
}

1  root = new Node(0, ∞1, {0.0}^d, null, null, null);
2  root->right = new Node(0, 0.0, {∞3}^d, null, null, null);
3  root->left = new Node(0, ∞2, {0.0}^d, null, null, null);
4  root->left->left = new Node(0, 0.0, {∞4}^d, null, null, null);
5  root->left->right = new Node(0, 0.0, {∞5}^d, null, null, null);

   Search(search_field& f, double k[DIMENSION])
6    f.ac = root;
7    f.sc = root->lt;
8    f.sc_dir = true;
9    f.pr = f.sc;
10   f.cr = f.pr->lt
11   f.cr_dir = true;
12   while not (f.cr->isLeaf())
13     if not (IsTag(f.pr)) {
14       f.ac = f.pr;
15       f.sc = f.cr;
16       f.sc_dir = f.cr_dir;
17     f.pr = f.cr;
18     f.cr_dir = k[f.pr->d] < f.pr->c;
19     f.cr = f.pr->GetChild(f.cr_dir);

   Contains(double k[DIMENSION])
20   start_op();
21   search_field f;
22   Search(f, k);
23   if not (IsMark(f.cr))
24     if (isEqual(k, f.cr->k))
25       Sync(f.pr, f.cr);
26       end_op(); return true;
27   end_op(); return false;

   Insert(double k[DIMENSION])
28   start_op();
29   search_field f;
30   while (true) {
31     Search(f, k);
32     if not (isEqual(k, f.cr->k)) {
33       Node* new_internal = new_node();
34       Node* new_leaf = new_node();
35       new_leaf->SetKey(k);
36       new_leaf->pr = new_internal;
37       int dim = GetMinDiffDimension(new_leaf->k, f.cr->k);
38       new_internal->d = dim;
39       new_internal->c = (new_leaf->k[dim] + f.cr->k[dim]) / 2;
40       new_internal->pr = f.pr;
41       bool new_leaf_dir = new_leaf->k [dim] < f.cr->k[dim];
42       if (new_leaf_dir) {
43         new_internal->lt = new_leaf;
44         new_internal->rt = f.cr;
45       else
46         new_internal->lt = f.cr;
47         new_internal->rt = new_leaf;
48       if (f.pr->CAS_Child(f.cr, null, new_internal, null, f.cr_dir))
49         CAS(&f.cr->pr, f.pr, new_internal);
50         f.pr = new_internal;
51         f.cr = new_leaf;
52         f.cr_dir = new_leaf_dir;
53         Sync(f.pr, f.cr);
54         end_op(); return true;
55       else {
56         del_node(new_leaf);
57         del_node(new_internal);
58         Node* child = f.pr->GetChild(f.cr_dir);
59         if ((child==f.cr) && (IsMark(child) || IsTag(child)) )
60           Cleanup(f);
61     else end_op(); return false;

enum RemoveMode { INJECTION, CLEANUP };
   Remove(double k[DIMENSION])
62   start_op();
63   search_field f;
64   RemoveMode mode = INJECTION;
65   Node* del = null;
66   while (true) {
67     Search(f, k);
68     if (INJECTION == mode)
69       if not (isEqual(k, f.cr->k))
70         end_op(); return false;
71       del = f.cr;
72       if (f.pr->CAS_Child(f.cr, null, f.cr, MARK, f.cr_dir))
73         mode = CLEANUP;
74         if (Cleanup(f)) end_op(); return true;
75       else
76         Node* child = f.pr->GetChild(f.cr_dir);
77         if ((child==f.cr) && (IsMark(child) || IsTag(child)) )
78           Cleanup(f);
79     else
80       if (del != f.cr)
81         end_op(); return true;
82       else if (Cleanup(f))
83         end_op(); return true;

   Cleanup(search_field& f)
84   Node* child = f.pr->GetChild(f.cr_dir);
85   Node* sibling = f.pr->GetChild(!f.cr_dir);
86   if not (IsMark(child))
87     sibling = child;
88     f.cr_dir = !f.cr_dir;
89   if not (IsTag(sibling))
90     f.pr->CAS_Child(sibling, LSB(sibling), sibling,
       LSB(sibling)|TAG, !f.cr_dir);
91   sibling = f.pr->GetChild(!f.cr_dir);
92   if not (IsTag(sibling)) return false;
93   if (f.ac->CAS_Child(f.sc, null, LSB(sibling),
       LSB(sibling)&MARK, f.sc_dir))
94     CAS(&sibling->pr, f.pr, f.ac);
95     retire_subtree(f.sc, sibling);
96     return true;
97   return false;

   CAS_Child(Node* old, ptrdiff_t old_lsb, Node* new, ptrdiff_t
     new_lsb, bool ch_dir)
     ptrdiff_t old_ptr = old | old_lsb;
     ptrdiff_t new_ptr = new | new_lsb;
     return (ch_dir) ? CAS(&lt, old_ptr, new_ptr) : CAS(&rt, old_ptr, new_ptr);
```

[Fig. 9] The structures and Read/Write Operations in HD-Tree

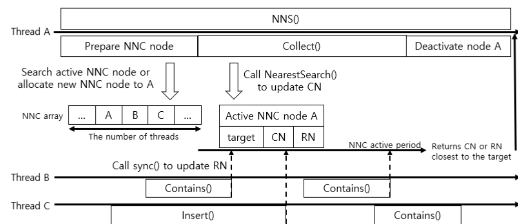
재할 경우 후처리를 시도한다.

Remove()의 68-78라인 INJECTION단계에서는 Search()를 통해 찾은 노드가 제거할 노드인지 확인하고 CAS를 통해 논리적인 제거를 시도한다. CAS가 실패할 경우 Insert()와 마찬가지로 후처리를 시도한 후 검색부터 다시 시도하고, 성공할 경우 80-83라인의 후처리 단계에서 Cleanup()를 시도하여 성공하거나 같은 키 값으로 검색했을 때 Marking된 노드가 더 이상 검색되지 않을 경우 true를 반환한다.

Cleanup()의 84-90라인에서는 후처리를 할 리프 노드에 Tag가 표시되어 있는지 확인하고, Tag가 없다면 Tag를 LSB에 삽입한다. 92라인에서는 Tag가 제대로 삽입되었는지 확인하여 삽입이 실패했을 경우 후처리를 종료하고, 성공했을 경우 물리적인 제거를 시도한다.

3.2 NNS의 Lock-Free구현

HD-Tree는 [Fig. 10]에서 보여주는 것처럼, 기존 알고리즘에서 NNS를 한 번 실행할 때 NNC는 하나만 사용되므로 모든 스레드가 NNS를 동시에 실행한다고 하더라도 사용되는 NNC노드의 수는 스레드의 수를 넘지 않는다는 아이디어를 통해 개선되었다.



[Fig. 10] NNS in HD-Tree

HD-Tree의 NNS알고리즘은 다음과 같다.

1) NNC배열에서 활성화된 NNC중 주어진 키 값과 target 값이 같은 NNC를 찾는다.

2) NNC를 찾는데 실패했을 경우, 사용할 NNC를 생성하고 주어진 키 값을 target 값으로 초기화한다. 그리고 생성한 NNC를 NNC배열의 스레드id 위치에 CAS를 통해 대입하며 기존의 비활성화된

NNC를 제거한다.

3) 1), 2)를 통해 얻은 NNC를 사용하여 2.2절에서 설명한 NNS알고리즘을 통해 CN을 얻는다.

4) 사용한 NNC를 비활성화하고 CN과 RN중 target에 더 가까운 노드를 반환한다.

개선된 NNS는 NNC리스트 대신 스레드의 수만큼의 크기를 가지는 NNC의 고정크기 배열을 사용하여 활성화된 노드를 찾는 검색시간을 상수 시간으로 제한시켜 성능을 향상시켰으며, NNS를 끝마쳤을 때 NNC노드를 제거하지 않고 새로운 노드를 생성해서 배열에 CAS를 통해 추가하는 경우에 이전 노드를 제거하여 기존 알고리즘이 가진 문제를 해결하였다.

[Fig. 11]은 HD-Tree의 NNS에서 사용되는 함수들의 수도코드이다. 여기에 사용되는 구조체들은 [Fig. 9]에 정의되어있다. NNS에서 사용되는 Neighbor에서 p는 NNS를 통해 찾은 노드이고 d는 NNCollector의 target(tgt)과 p의 키 값과의 거리를 나타낸다. NNCollector에서 tgt는 NNS()의 인자로 주어진 키 값이고, cn은 Collected Neighbor의 약자이며 NNS()를 호출한 스레드가 Collect()를 통해 얻은 Nearest Neighbor이다. rn은 Reported Neighbor의 약자이고 NNS에 사용되고 있는 NNC가 활성화 되어있는 동안, 다른 스레드들이 Insert()나 Contains()를 성공하여 호출한 Sync()를 통해 갱신한다.

seek_field는 NN을 찾는데 사용된다. cr, pr, cr_dir는 검색을 통해 찾은 리프노드와 부모노드, 리프노드의 방향을 의미하며, lo와 hi는 KD-Tree의 검색을 통해 얻은 cr이 위치한 bounding box를 나타낸다. NeighborSearch()에서 bounding box는 NN을 찾기 위해 반복검색을 할 때 한번 방문했던 영역을 중복방문하지 않기 위해서 사용된다.

1-3라인은 NNS에서 사용되는 NNC고정크기 배열을 알고리즘을 시작하기 전에 초기화 하는 것을

```
enum NeighborType { COLLECTED, REPORTED };
1  for (int i = 0; i < MAX_THREAD; ++i)
2    nncs[i] = new NNCollector();
3    Deactivate(nncs[i]);

NNS(double k[DIMENSION])
4  start_op();
5  seek_field f;
6  f.pr = root;
7  f.cr = root->lt;
8  f.cr_dir = true;
9  f.hi = {∞}^d;
10 f.lo = {−∞}^d;
11 Seek(f, k);
12 double dst = ∞;
13 if not (IsMark(f.cr))
14   dst = getdst(k, f.cr->k);
15 if not (isZero(dst))
16   ret = NNSync(f, dst, k)->k;
17   end_op(); return ret;
18 else
19   Sync(f.pr, f.cr);
20   end_op(); return k;

CheckValid(Node* pr, Node* cr)
  bool cr_dir = cr->k[pr->d] < pr->c;
  Node* child = pr->GetChild(cr_dir);
  while not (child->isLeaf())
    cr_dir = cr->k[child->d] < child->c;
    child = child->GetChild(cr_dir);
  if (IsMark(child)) return false;
  return child == cr;

Seek(seek_field& f, double k[DIMENSION])
21 while not (f.cr->isLeaf())
22   f.pr = f.cr;
23   f.cr_dir = k[f.pr->d] < f.pr->c;
24   f.cr = f.pr->GetChild(f.cr_dir);
25   if (f.cr_dir)
26     f.hi[f.pr->d] = f.pr->c;
27   else
28     f.lo[f.pr->d] = f.pr->c;

Sync(Node* pr, Node* cr)
29 for (int i = 0; i < MAX_THREAD; ++i)
30   if (IsActive(nncs[i])) {
31     if (IsNearNbr(nncs[i], getdst(cr->k, nncs[i]->tgt)))
32       if (CheckValid(pr, cr))
33         UpdateNbr(nncs[i], cr, REPORTED);
34       else break;

Deactivate(NNCollector* nnc)
  NNCollector* old_nnc = nnc;
  while not (CAS(&nnc, old_nnc, old_nnc|DEACTIVATION))
    old_nnc = nnc;

enum NNSyncMode { INIT, COLLECT };
NNSync(seek_field& f, double dst, double k[DIMENSION])
35 NNCollector* nnc = null;
36 NNSyncMode mode = INIT;
37 while (true) {
38 for (int i = 0; i < MAX_THREAD; ++i)
39   if (IsActive(nncs[i]))
40     if (isEqual(k, nncs[i]->tgt))
41       nnc = nncs[i];
42       mode = COLLECT;
43       break;
44 if (INIT == mode)
45   nnc = new_nnc(k, new_nbr(f.cr, dst), new_nbr(f.cr, dst));
46   NNCollector* old_nnc = nncs[tid];
47   while (false == CAS(&nncs[tid], old_nnc|DEACTIVATION, nnc))
48     old_nnc = nncs[tid];
49   retire(old_nnc->cn);
50   retire(old_nnc->rn);
51   retire(old_nnc);
52   mode = COLLECT;
53 if (COLLECT == mode) {
54   nnc = Collect(f, dst, k, nnc);
55   Deactivate(nnc);
56   return Process(nnc);

Collect(seek_field& f, double dst, double k[DIMENSION],
    NNCollector* nnc)
57 while ((root != f.pr) && (false == isZero(dst)))
58   NeighborSearch(f, dst, k);
59   if (CheckValid(f.pr, f.cr))
60     dst = UpdateNbr(nnc, f.cr, COLLECTED);
61 return nnc;

NeighborSearch(seek_field& f, double dst, double
    k[DIMENSION])
62 double leaf_key[] = f.cr->k;
63 while (root != f.pr) {
64   bool bVisited = false;
65   if (f.cr_dir) {
66     bVisited = f.pr->c >= f.hi[f.pr->d];
67   else
68     bVisited = f.pr->c <= f.lo[f.pr->d];
69   if (bVisited && (abs(f.pr->c - k[f.pr->d]) < dst))
70     f.cr_dir = !f.cr_dir;
71     f.cr = f.pr->GetChild(f.cr_dir);
72     Seek(f, k);
73     leaf_key = f.cr->k;
74     double leaf_dst = getdst(k, leaf_key);
75     if (leaf_dst < dst)
76       if not (IsMark(f.cr))
77         dst = leaf_dst;
78         break;
79   else
80     f.cr = f.pr;
81     f.pr = f.pr->pr;
82     f.cr_dir = f.pr->GetChildDir(leaf_key);
83     if (f.cr_dir)
84       if (f.pr->c > f.hi[f.pr->d]) f.hi[f.pr->d] = f.pr->c;
85     else
86       if (f.pr->c < f.lo[f.pr->d]) f.lo[f.pr->d] = f.pr->c;

UpdateNbr(NNCollector* nnc, Node* cr,
    NeighborType nt)
87  double dst = getdst(cr->k, nnc->tgt);
88  while (true)
89    if (IsActive(nnc))
90      if (true == IsNearNbr(nnc, dst))
91        Neighbor* nbr = new_nbr(cr, dst);
92        Neighbor* old_nbr = null;
93        bool result = false;
94        if (COLLECTED == nt)
95          old_nbr = nnc->cn;
96          result = CAS(&nnc->cn, old_nbr, nbr);
97        else
98          old_nbr = nnc->rn;
99          result = CAS(&nnc->rn, old_nbr, nbr);
100       if (result)
101         retire(old_nbr);
102         return dst;
103       else del_nbr(nbr);
104   else return dst;

IsNearNbr(NNCollector* nnc, double target_dst)
  return (target_dst < nnc->cn->d
      && target_dst < nnc->rn->d);

Process(NNCollector* nnc)
  return (nnc->rn->d < nnc->cn->d)
      ? nnc->rn->p : nnc->cn->p;
```

[Fig. 11] NNS Operations in HD-Tree

나타낸다. 5-10라인은 seek_field를 초기화하여 11라인에서 seek()을 통해 주어진 키 값에 대한 검색결과를 얻는다. 13라인에서 이렇게 얻은 리프노드의 LSB를 확인하여 유효한지 확인하고, 키 값과 주어진 키 값 사이의 거리가 0이라면 KD-Tree내부에 찾는 위치에 겹쳐있는 노드가 존재하는 것이므로 추가적인 작업을 하지 않고 리프노드의 키 값을 반환한다.

리프노드가 유효하지 않거나 주어진 키 값 사이의 거리가 0이 아니라면 NNSync()를 실행한다. 38-43라인에서는 배열 순회를 통해 활성화되어 있으면서 주어진 키 값과 같은 target을 가지고 있는 NNC를 찾는다. 44-52라인에서는 적합한 노드를 찾는데 실패했을 경우 새로운 NNC노드를 생성하여 현재 NNS를 진행하고 있는 스레드 자신의 id인 tid를 배열의 인덱스로 사용하여 NNC배열에 CAS를 통해 추가하고, 배열에 들어있던 이전 NNC노드를 EBR알고리즘을 통해 제거한다. 53-56라인에서는, NNS에서 사용할 NNC를 얻은 후 Collect()를 진행하고 Collect()가 반환되었을 경우 사용한 NNC를 Deactivate()를 통해 비활성화 하고 Process()에서 cn과 rn 중 target에 더 가까운 노드를 반환한다.

UpdateNbr()는 새로 찾은 노드가 Neighbor에 저장된 노드보다 target 값에 더 가까운지 IsNearNbr()를 통해 확인한 후 갱신하고, 갱신하기 전 저장되어있던 노드를 EBR알고리즘을 통해 제거한다.

CheckValid()는 인자로 주어지는 리프노드가 부모로부터 도달할 수 있는지, LSB가 삽입되지 않았는지 검사하여 리프노드의 유효성을 검사한다.

Collect()는 2장 2절에서 설명했었던 NNS알고리즘을 수행한다. NeighborSearch()를 통해 KD-Tree를 한번 탐색하여 얻은 리프노드가 유효한지 CheckValid()를 통해 확인한 후 UpdateNbr()를 사용하여 cn을 갱신한다. 이 과정은 NeighborSearch()가 더 이상 주어진 목표 키 값에서 탐색 가능한 가까운 영역을 발견하지 못할 때 까지 반복된다.

4. 실험 및 평가

HD-Tree의 실험은 AMD Ryzen 7 2700X 3.7GHz의 8 코어 16 스레드, 16G RAM 환경에서 진행되었으며, 다음 세 가지의 알고리즘을 비교했다.

1) LFKD-Tree_JAVA: 기존의 JAVA알고리즘

2) LFKD-Tree_C++: 기존의 JAVA알고리즘에 EBR을 추가하여 C++로 구현

3) HD-Tree: 본 논문에서 제시한 알고리즘

LFKD-Tree_C++를 구현할 때 NNS에서 사용한 후 비활성화된 NNC노드를 후처리하지 않으면 비교가 불가능할 정도로 성능이 떨어지기 때문에 NNC노드를 후처리하는 루틴을 추가하여 알고리즘의 문제를 해결한 상태로 성능을 비교하였다.

실험은 각 스레드가 임의의 2차원 점을 키 값으로 사용하여 2초 동안 실행하는 것을 5번씩 반복하여 실행시간의 평균값으로 초당 실행된 작업 수를 계산해서 비교하였다.

실험의 구성은 다음과 같다.

1) 스레드 수 ∈ { 1, 2, 4, 8, 16 }

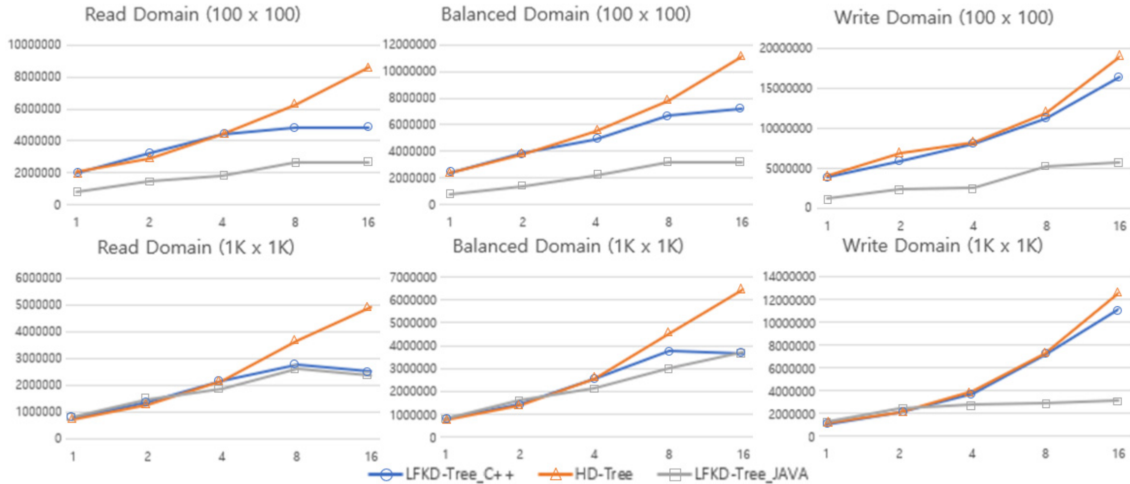
2) Key space ∈{ (100 x 100), (1K x 1K) }

3) 함수 실행비율 (NNS-Insert-Remove)

- Read Domain: (80-10-10)
- Balanced Domain: (50-25-25)
- Write Domain: (0-50-50)

삽입/삭제 비율이 낮고 NNS비율이 높은 Read Domain은 NNS의 성능을 측정하기 용이하고 NNS를 실행하지 않는 Write Domain은 삽입/삭제 알고리즘의 성능을 측정할 때 사용할 수 있다. Balanced Domain은 Read/Write Domain와 함께 비교하여 NNS/삽입/삭제 알고리즘이 전체 성능에 얼마나 영향을 미치는지 알 수 있다.

기존 JAVA로 구현된 알고리즘을 EBR을 적용하여 C++로 변환한 결과 삽입/삭제의 성능이 최대 252%만큼 좋아졌고, NNS의 성능이 최대 79%만



[Fig. 12] Result of the experiment

큼 좋아졌다.

기존 알고리즘은 노드 제거 후 후처리를 할 때 한 번에 하나의 노드만 제거할 수 있어서 여러 번 후처리를 해야 하는 경우가 있지만, HD-Tree는 후처리가 필요한 노드들이 연결되어 있을 경우 한 번에 처리할 수 있다. 그 결과 HD-Tree의 경우 하드웨어 코어 수보다 스레드 수가 많은 상황에서 성능을 최대 15% 만큼 향상시킬 수 있었다.

NNS에서 Key space가 1K x 1K일 경우 관리하는 노드 수가 늘어나서 메모리를 관리하는 EBR 알고리즘의 오버헤드가 커지기 때문에 스레드 수가 적을 때 HD-Tree가 JAVA보다 성능이 떨어지는 경우가 생기지만, 기존 알고리즘이 가지고 있던 문제점을 해결했기 때문에 스레드 수가 늘어나는 만큼 성능이 향상되는 것을 확인할 수 있다.

동기화 방법을 개선한 HD-Tree의 NNS는 기존 알고리즘의 C++구현보다 성능이 최대 95%만큼 좋아졌다.

5. 결론 및 향후연구

본 논문에서는 기존의 Lock-Free KD-Tree의 동기화 방법을 수정해서 성능을 개선한 HD-Tree를 제안하였다. 기존 NNS알고리즘에서 Lock-Free 리스트를 사용하여 노드를 관리하는 부분을 총 스레드 개수만큼의 고정크기 배열로 변경하는 것으로 지나친 리스트 순회로 성능이 떨어지던 문제를 해결하여 높은 성능을 얻었고, 트리의 노드를 제거하는 부분에서 사용하는 CAS의 수를 줄이고 연결된 제거가 필요한 노드들을 한 번에 처리할 수 있도록 변경하여 높은 경합 상황에서의 성능이 향상되었다.

향후 연구과제는 동기화 알고리즘을 더욱 최적화하거나 병행 NNS알고리즘을 더욱 발전시켜 KD-Tree 내부의 기준거리 미만의 노드들을 수집하여 반환하는 연산과 같은 게임개발에 유용한 기능을 추가하는 것이다.

REFERENCES

[1] B. Chatterjee, I. Walulya, and P. Tsigas. “Concurrent linearizable nearest neighbour search in lockfree-kd-tree”. ICDCN '18: Proceedings of the 19th International Conference on Distributed Computing and Networking, 2018.

[2] A. Natarajan, A. Ramachandran and N. Mittal. “FEAST: A Lightweight Lock-free Concurrent Binary Search Tree”. ACM Transactions on Parallel Computing. 2020.

[3] H. Wen, J. Izraelevitz, W. Cai, H. A. Beadle and M. L. Scott. “Interval-based memory reclamation”. ACM SIGPLAN Notices. 2018.
[4] Steam Hardware&Software Survey: July 2020. https://store.steampowered.com/hwsurvey/cpus/
[5] M. Herlihy and N. Shavit. “The Art of Multiprocessor Programming Revised Reprint”. Morgan Kaufmann, 2012.
[6] J. L. Bentley. “Multidimensional binary search trees used for associative searching”. CACM, vol. 18, no. 9, pp. 509-517, 1975.
[7] M. Herlihy. “Wait-Free Synchronization”. ACM Transactions on Programming Languages and Systems (TOPLAS), 13 (1): 124-149, Jan. 1991.
[8] D. Dechev, P. Pirkelbauer and B. Stroustrup. “Understanding and Effectively Preventing the ABA Problem in Descriptor-based Lock-free Designs”. 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, 2010.
[9] M. Michael, “Hazard pointers: Safe memory reclamation for lock-free objects”, IEEE Transactions on Parallel and Distributed Systems, 15 (6): 491 - 504. 2004.
[10] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel. “Non-blocking binary search trees”. In Proceedings of the 29th ACM Symposium on Principles of Distributed Computing (PODC’10). ACM, New York, NY, 131 - 140. 2010.
[11] T. Brown, F. Ellen, and E. Ruppert. “A general technique for non-blocking trees”. In Proceedings of the 19th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP’14). ACM, New York, NY, 329 - 342. 2014.

이 상 기 (Lee, Sang-gi)

약 력 : 2019 한국산업기술대학교 게임공학과 학사
2019-현재 한국산업기술대학교 일반대학원 디지털엔터테인먼트학과 석사과정

관심분야 : 병렬처리, 컴퓨터 그래픽스, 게임엔진, 게임서버

정 내 훈 (Jung, NaiHoon)

약 력 : 2002 KAIST 전산학과 박사
2002-2008 NCSoft MMORPG 프로그램팀장
2008-현재 한국산업기술대학교 게임공학부 부교수

관심분야 : 병렬처리, 컴퓨터 구조, 대용량 온라인 게임 서버