**Data Structure**

**~ 과제 5 Integer Application ~**

2013210111 남세현

1. 목표

Binary Search Tree 와 Linear Lists(Linked List)에 integer 값들을 insert, delete, search를 해보고 performance를 비교합니다.

1. 실행화면

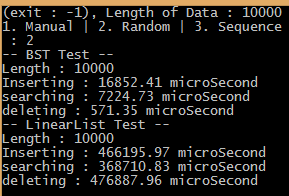


Figure 1 : 1만개의 Random 데이터 테스트

1. 구현 방법

**[ Linear List ]**

한 Node가 다음 Node를 가리킬 수 있는 nextNode 포인터를 가진 형태의 데이터 구조입니다.



1. struct LinearListNode\* LinearListNode\_Create(int value);

Linear List의 Node를 생성하여 pointer를 반환하는 함수입니다.

struct LinearList\* LinearList\_Craete();

Linear list를 생성하는 함수입니다.

1. void LinearList\_Destroy(struct LinearList\* list);

생성했던 Linear List를 제거하는 함수입니다

* 순차적으로 List 내부를 Iteration하면서 모든 node를 free해줍니다.

1. void LinearList\_Insert(struct LinearList\* list, int value);

Linear List에 값을 추가하는 함수입니다.

* iteration으로 제일 마지막 node를 찾은 후, LinearListNode\_Create로 노드를 하나 생성 후 값을 넣어줍니다.

1. struct LinearListNode\* LinearList\_Search(struct LinearList\* list, int value);

value를 가지고 있는 Node를 찾는 함수입니다.

* Iteration을 수행하면서 value를 가지고 있는 node를 찾고, 포인터로 반환합니다.

1. void LinearList\_Delete(struct LinearList\* list, int value);

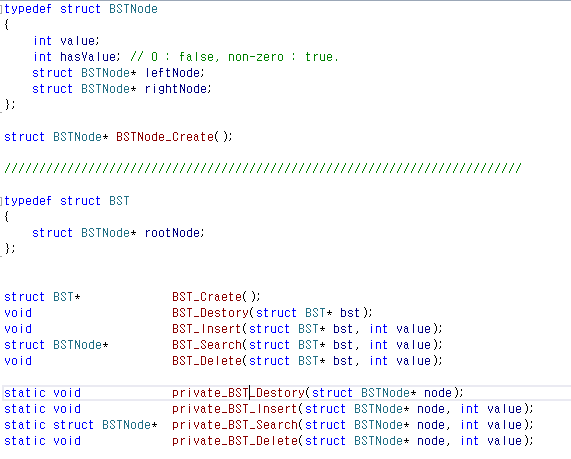
value를 가지고 있는 node를 제거하는 함수입니다.

* Search와 비슷하나, 포인터 반환이 아니라 free를 해줍니다.

**[ BST ]**

Linear List가 오직 하나의 next Node를 가졌다면, BST에서는 왼쪽과 오른쪽, 총 2개의 Next Node를 가지게 됩니다.

또한 개발 편의를 위해서 hasValue라는 boolean값을 가지고 있습니다.



1. struct BSTNode\* BSTNode\_Create();

BST 용 Node를 생성해서 포인터를 반환합니다.

1. struct BST\* BST\_Craete();

BST를 생성하고 포인터를 반환합니다.

1. void BST\_Destory(struct BST\* bst);

BST를 제거합니다.

* 모든 Node들을 iteration하면서 free해줍니다.
* Iteration은 recursive function을 이용하여 접근하였습니다.

1. void BST\_Insert(struct BST\* bst, int value);

BST에 value를 가진 노드를 추가합니다.

* static void private\_BST\_Insert(struct BSTNode\* node, int value); 함수로 recursive하게 처리합니다.
* 현재 node가 hasValue == false면 node에 value를 넣어줍니다.
* 현재 node가 hasValue == true면, 현재 값(node->value)와 value를 비교합니다.
  + 비교했을 때 value가 작았다면 left Node를,
  + Value가 컸다면 right Node를 재귀 호출합니다.
  + 만약 next node 자체가 존재하지 않다면 BSTNode\_Create로 노드 자체를 추가한 후 재귀합니다.

1. struct BSTNode\* BST\_Search(struct BST\* bst, int value);

BST에 value를 가진 노드를 찾아냅니다.

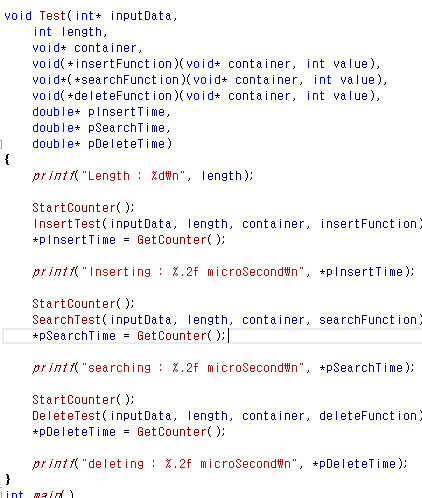
* Iteration을 돌면서 node->value == value를 만족하는 node를 찾아냅니다.
* 만약 현재 node->value != value라면,
  + 비교했을 때 value가 작았다면 left Node를,
  + Value가 컸다면 right Node를 재귀 호출합니다.
* 만약 해당 값을 가진 value를 찾기 전에 node == NULL이라면, NULL를 반환합니다.

1. void BST\_Delete(struct BST\* bst, int value);

BST에 value를 가진 노드를 찾아내 지웁니다.

* Iteration을 돌면서 node->value == value를 만족하는 node를 찾아냅니다.
* 만약 현재 node->value != value라면,
  + 비교했을 때 value가 작았다면 left Node를,
  + Value가 컸다면 right Node를 재귀 호출합니다.
* 만약 해당 값을 가진 value를 찾기 전에 node == NULL이라면, 바로 종료합니다.
* **\* 값을찾아낸 경우, free를 하지 않고 hasValue 값을 false로만 만들어줍니다.**
  + **메모리 재활용을 위함.**

**[ TEST ]**



포인터, 함수포인터를 적절히 사용해서 구현했습니다.

1. 테스트할 데이터와 데이터 크기,
2. 테스트할 container(BST or List)
3. 테스트할 insert, search, delete 함수들
4. 테스트 결과를 반환할 double pointer

를 입력받아서 테스트를 수행합니다.

테스트 방법은 input Data를 전부 iteration하면서 해당 함수를 수행합니다.

**결과 분석**

결과 분석을 위해 다음과 같은 데이터를 넣어보았습니다.

1. 랜덤한 값 10, 100, 1000, 10000개

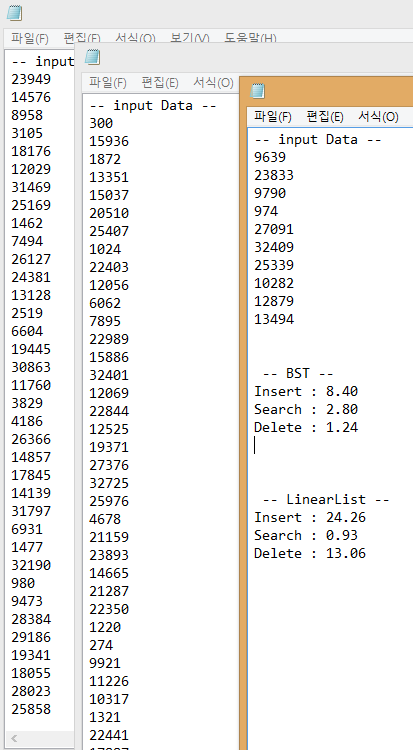


Figure 2 결과물. TEST 수행시 output.txt 파일이 만들어짐.

1. 랜덤한 값들을 넣었을 경우

|  |  |  |
| --- | --- | --- |
| 단위 : 마이크로세컨드 | BST | LinearList |
| 10개 | Insert : 7.46  Search : 0.31  Delete : 0.31 | Insert : 10.26  Search : 0.62  Delete : 8.40 |
| 100개 | Insert : 419.26  Search : 0.62  Delete : 0.62 | Insert : 1302.87  Search : 7.46  Delete : 181.01 |
| 1000개 | Insert : 9911.96  Search : 16.17  Delete : 2.49 | Insert : 4962.98  Search : 691.74  Delete : 3670.06 |
| 10000개 | Insert : 116372.56  Search : 24.26  Delete : 21.1 | Insert : 138523.23  Search : 105700.45  Delete : 342663.06 |

**분석**

Insert의 경우 BST는 nlogn에 가까운 값이 나왔는데, 총 n번의 연산의 결과이므로 최종적으론 insert 그 자체에 대해서는 O(n)에 유사하게 증가함을 볼 수 있었습니다.

하지만 Search나 Delete의 경우는 약간 다른데,

Delete의 경우 제가 BST에 memroy re-use 알고리즘을 추가했기 때문에 free연산이 없어서, Linear List에 비하면 굉장히 빠르게 동작함을 볼 수 있습니다.

현재는 각 state당 1번의 테스트를 진행했는데, 1 단계마다 n번의 연산을 수행하여 평균 값을 치루면 좀 더 이론적인 값에 가깝게 나올 수 있을 것 같습니다.

하지만 그렇지 않더라도 1개당 search 시간을 비교해보면

BST의 경우

|  |  |  |  |
| --- | --- | --- | --- |
| 데이터 10개 | 100개 | 1000개 | 10000개 |
| 0.28 | 0.45 | 0.36 | 0.71 |

linear하게 증가함을 볼 수 있는데, 즉 log(n)의 증가율을 보임을 볼 수 있습니다.

이론적으로 BST 의 Search는 O(log(n))인데, 유사한 값이 나옴을 확인할 수 있었습니다.