데이터 구조 설계 보고서

제목: DS_Project1

학　　과: 컴퓨터정보공학부

학　　번: 2022202064

성　　명: 최봉규

실습 분반: 금 5교시, 6교시

1. Introduction

이번 프로젝트는 이진 탐색 트리(Binary Search Tree), 링크드 리스트(Linked list), 큐(Queue) 자료 구조를 이용하여 개인정보 관리 프로그램을 구현한다. 이 프로그램은 데이터 파일로부터 회원이름, 나이, 개인정보수집일자, 가입약관종류 정보를 얻어 Queue를 구축하며, 해당 Queue를 Member_Queue라 부른다. 가입약관은 총 4 종류로 A, B, C, D가 있으며, 각 순서대로 6개월, 12개월 24개월 36개월의 개인정보 보관 유효기간을 갖는다. 각 Queue에서 pop 명령어를 실행하면 데이터를 방출하여 가입약관종류를 기준으로 정렬된 자료구조 (Terms_List, Terms_BST)와 회원이름을 기준으로 정렬된 자료구조(Name_BST)에 저장한다.

- 가입약관종류를 기준으로 정렬된 자료구조

Terms_List는 가입약관의 종류 별로 노드가 구성되며, 각 노드는 가입약관종류, 해당 가입약관의 회원 수, 가입약관의 BST 포인터 정보를 갖는다. 입력된 가입약관의 순서대로 노드가 생성되어 연결되며, 이미 존재하는 가입약관종류 정보가 입력될 경우 새로운 노드를 생성하지 않고 해당하는 기존 노드의 회원수를 증가시킨다. 정보가 입력된 후에는 BST 포인터에서 노드 구성을 수행한다.

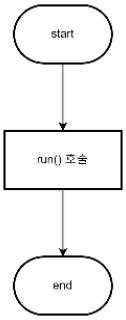
Terms_BST는 가입약관의 종류 별로 구성되며, 각 BST의 노드는 회원이름, 나이, 개인정보수집일자, 개인정보만료일자 정보를 갖는다. 개인정보만료일자는 개인정보수집일자에 가입약관 별 개인 정보 보관 유효기간이 더해져 계산되며, 각 BST는 개인정보만료일자 정보를 기준으로 정렬된다.

- 회원이름을 기준으로 정렬된 자료구조

Name_BST는 회원이름, 나이, 개인정보수집일자, 개인정보만료일자, 가입약관종류 정보를 갖는 노드로 구성되며, 회원이름 정보를 기준으로 정렬된다.

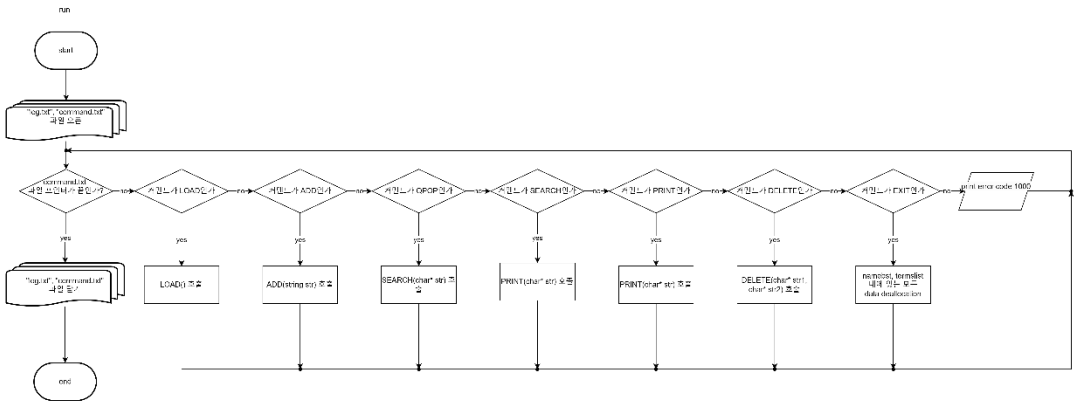
2. Flowchart

main.cpp

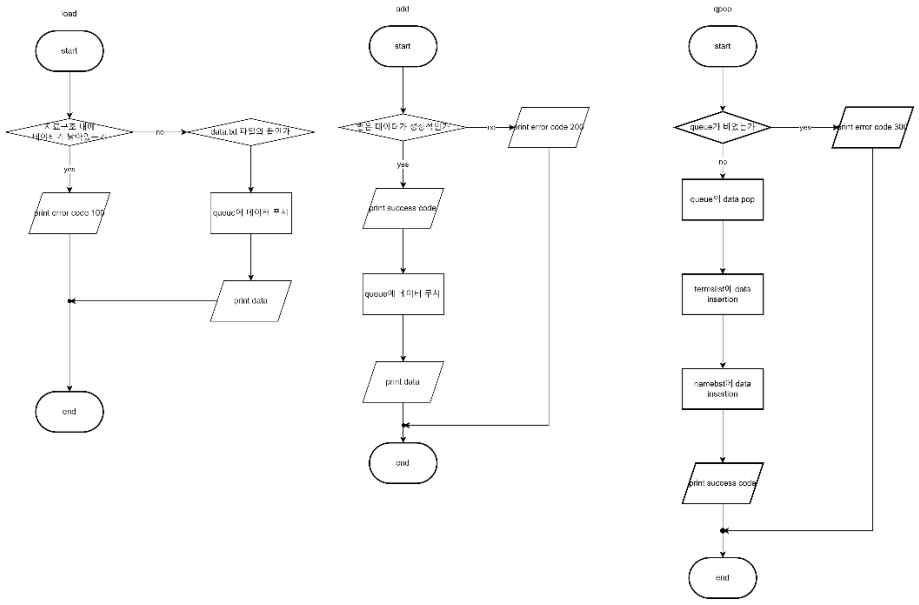


Main.cpp에 있는 동작을 설명

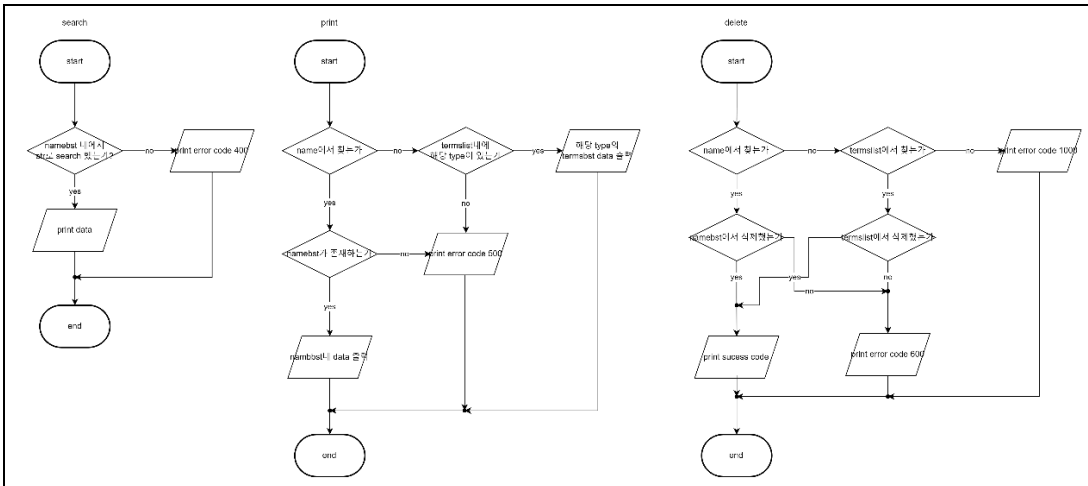
manager.cpp



이 순서도는 manager.cpp에 있는 일곱개의 커맨드에 따른 것이다. 명령어는 LOAD, ADD, QPOP, SEARCH, PRINT, DELETE, EXIT이 있다.

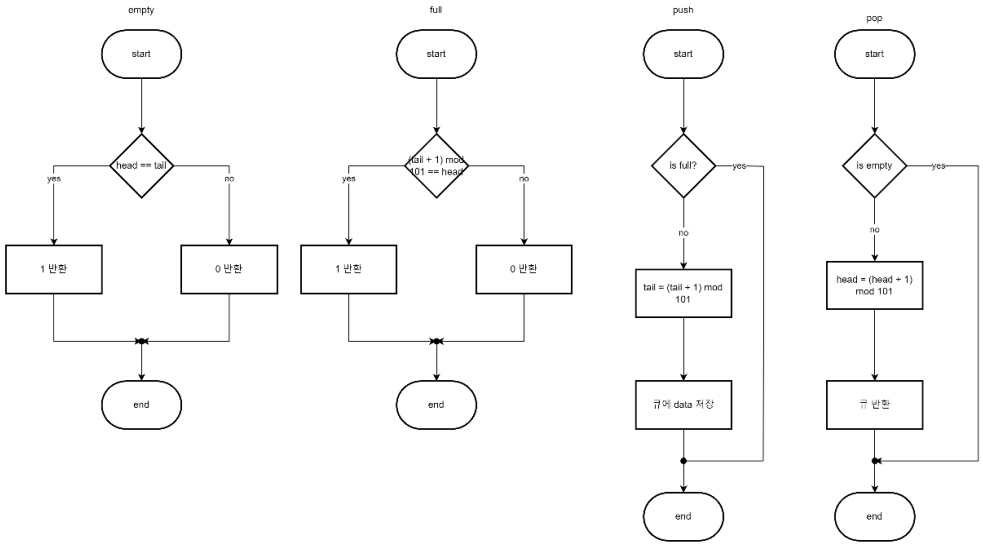


위의 세 개는 load, add, qpop에 대한 명령어 수행 연산 순서도이다.



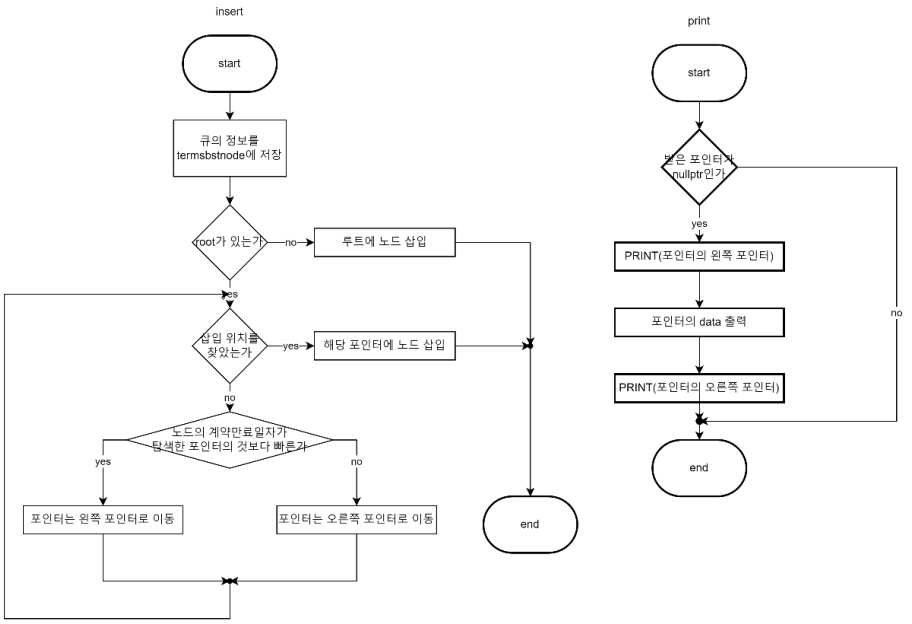
위의 세 개는 search, print, delete 에 대한 명령어 수행 연산 순서도이다.

MemberQueue.cpp

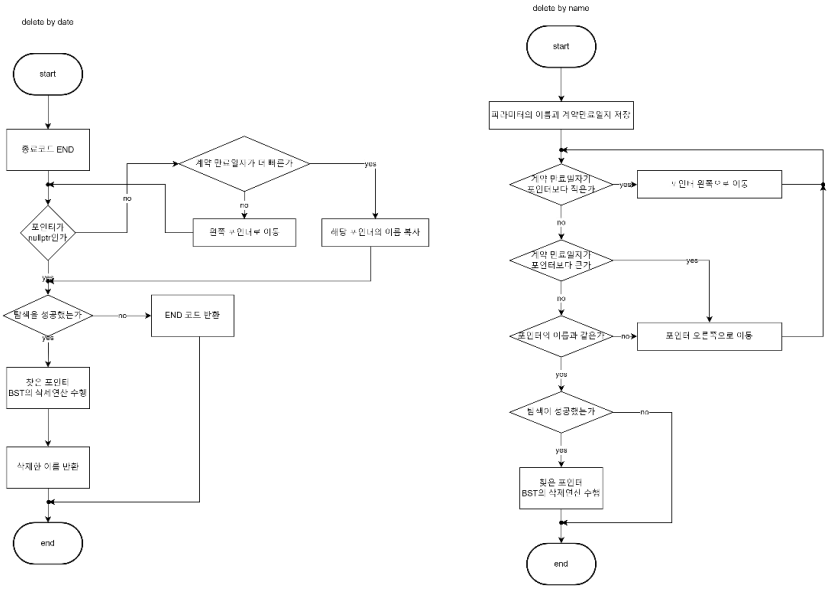


이 순서도에는 Memberqueue의 연산이 있다. Memberqueue에서의 연산은 empty, full, push, pop, front가 있다.

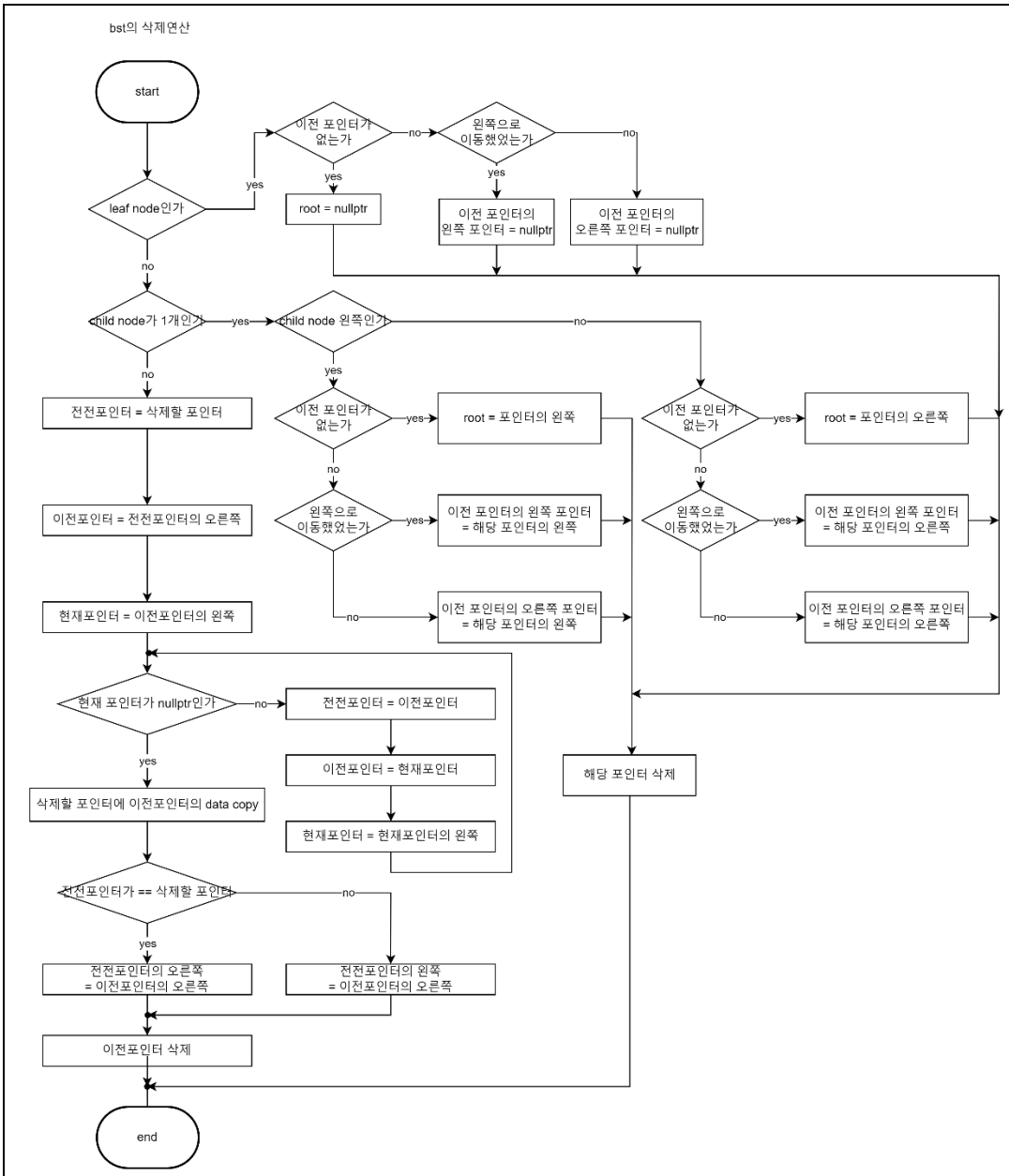
TermsBST.cpp



해당 순서도는 TermsBST.cpp에 있는 순서도 중 insert와 print에 대한 함수이다.

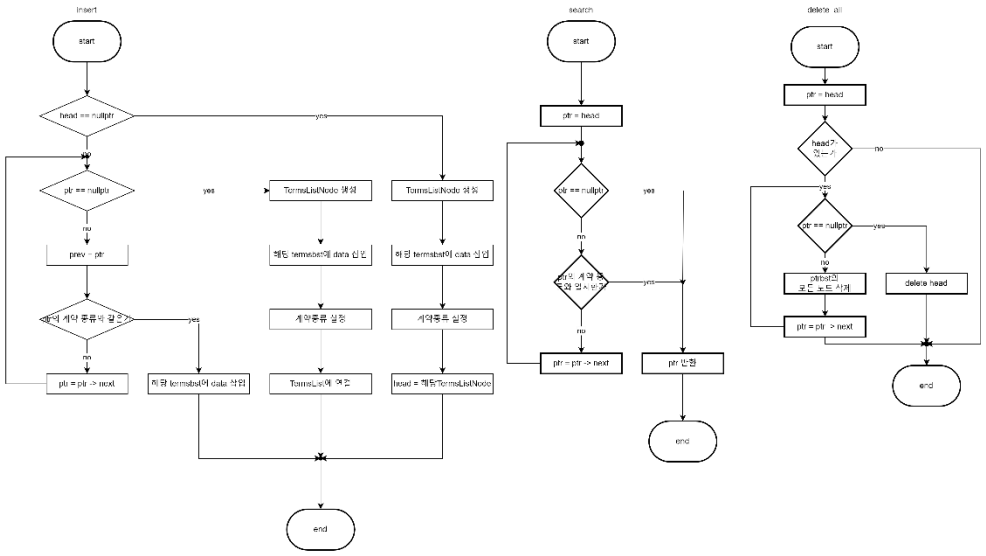


해당 순서도는 TermsBST.cpp에 있는 순서도 중 delete_by_date, delete_by_name가 있다. Delete_by_date는 계약 만료 기간을 인자로 받아 모든 node를 삭제하는 함수이다. Delete_by_name은 회원의 정보가 있는 memberqueuenode를 인자로 받아 해당 queue 노드의 이름을 가지고 탐색하며 삭제하는 함수이다.

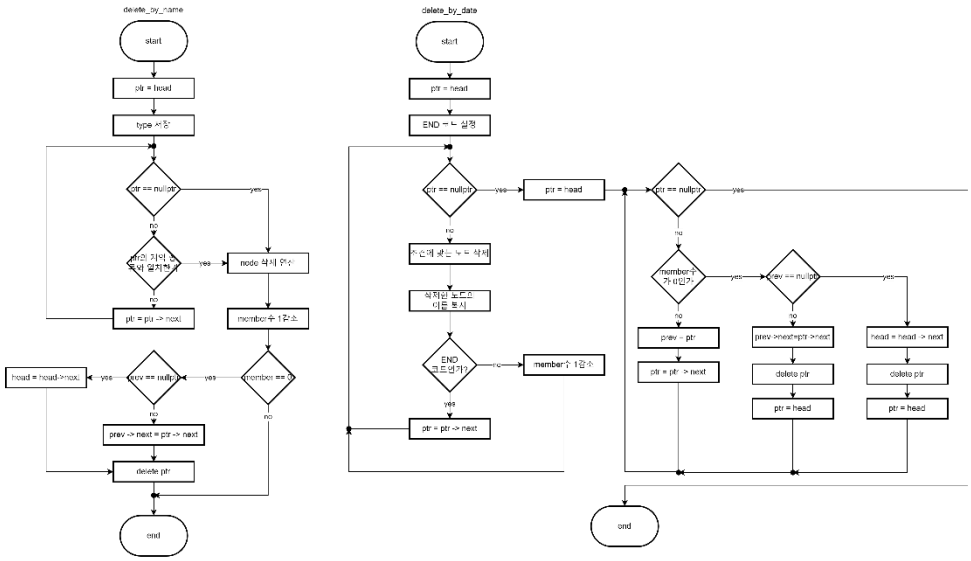


이 함수는 bst의 삭제연산을 나타내며 delete_by_name과 delete_by_date에서 해당 연산을 사용하나 데이터를 복사하는 차이는 있으나 일단 탐색을 하고 나서는 삭제하는 연산은 같기 때문에 하나로 보여준 순서도이다.

TermsList.cpp

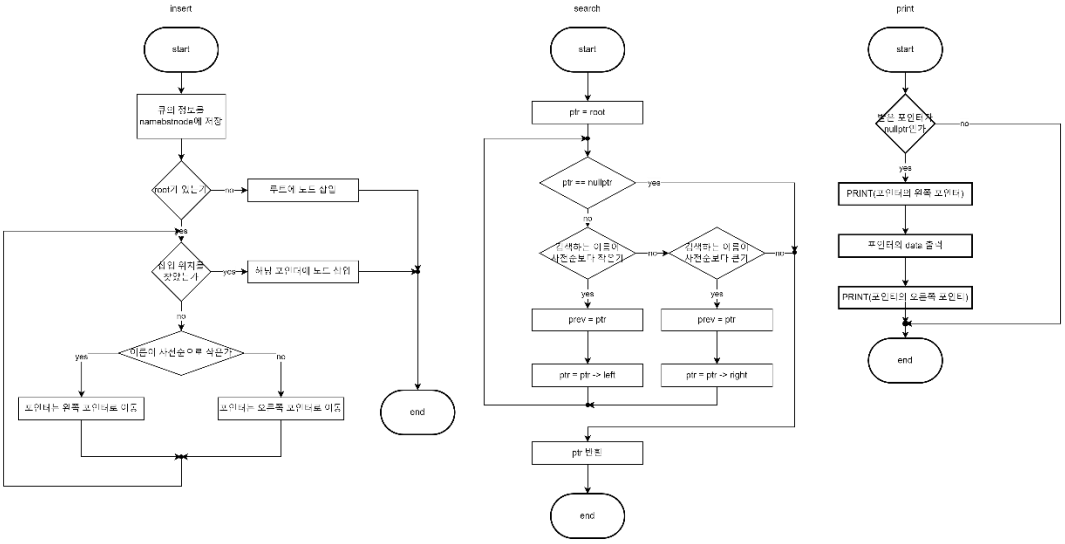


이 순서도들은 termslist에서의 insert, search, delete_all 연산을 보여주고 있다.

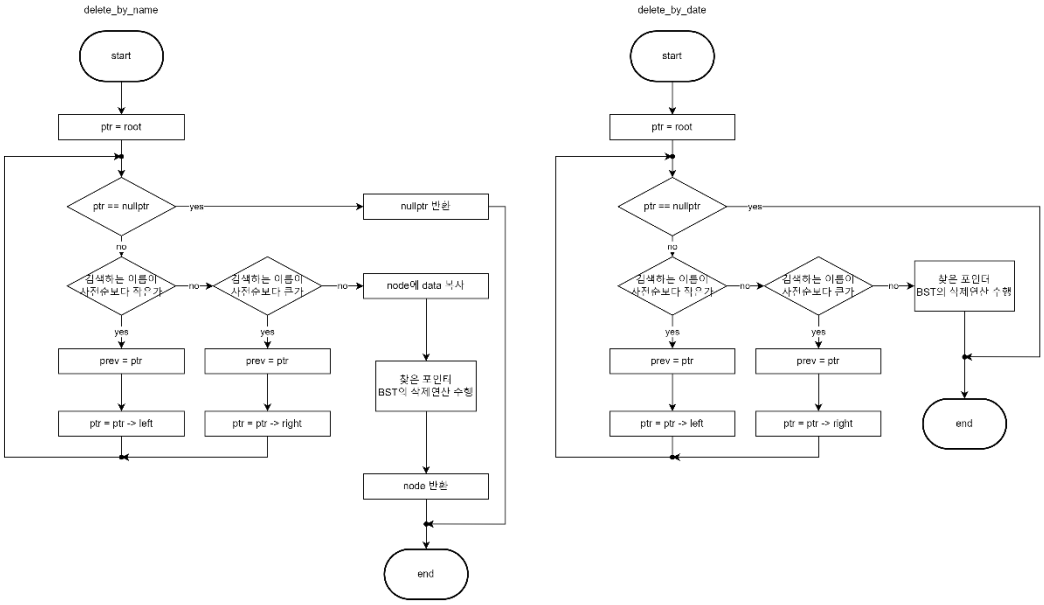


이 순서도들은 delete_by_name과 delete_by_terms에 대해 보여주고 있다. 해당 연산들을 통해 termbst.cpp에서의 삭제 연산을 한다.

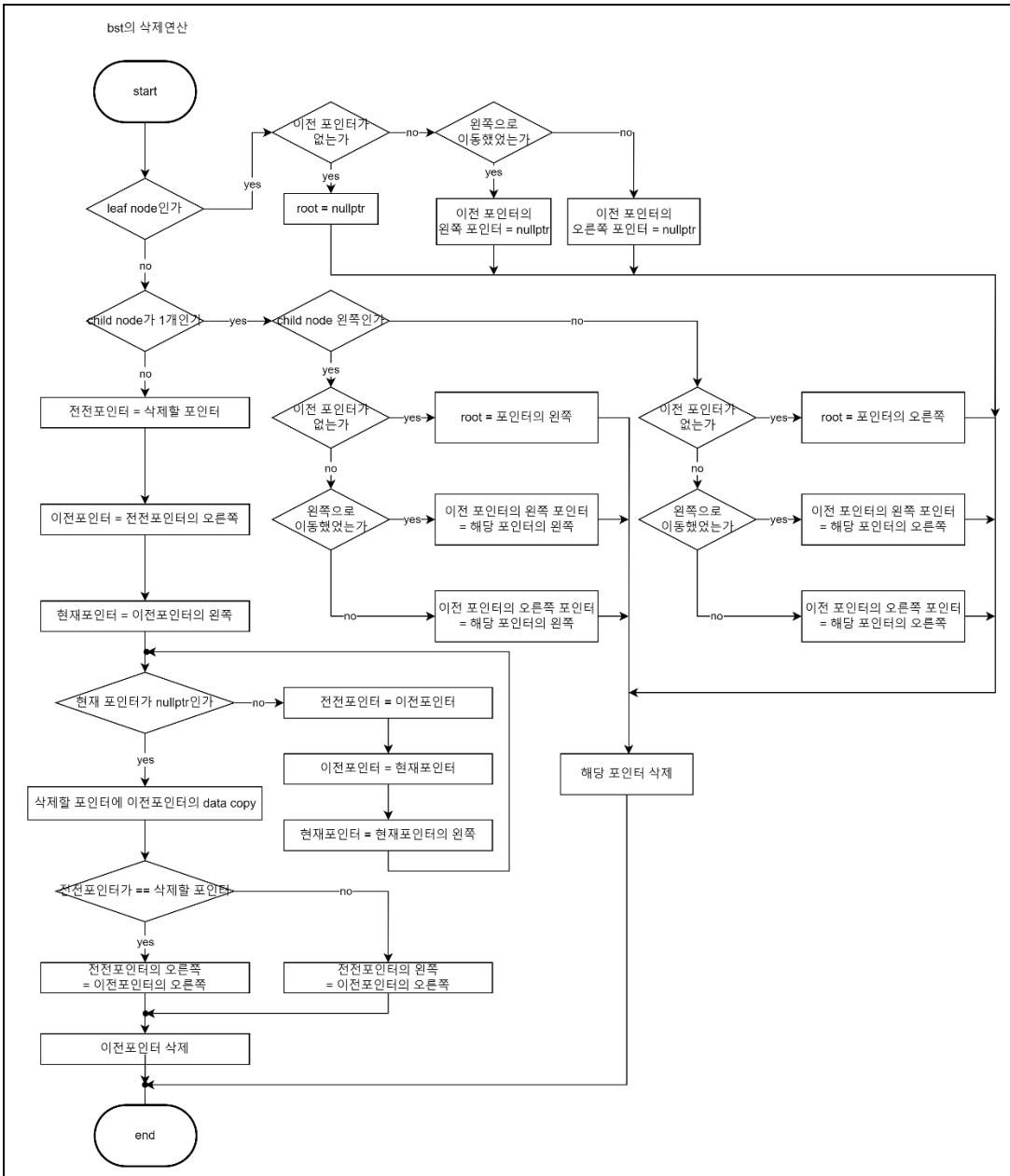
Namebst.cpp



위의 순서도 세개는 Namebst에서의 Insert함수와 Search, Print이다. 여기서 Print는 재귀적으로 움직인다.



Namebst ㅇ 역시 termslist와 termsbst와 마찬가지로 delete_by_name과 delete_by_date가 있다.



이 함수는 bst의 삭제연산을 나타내며 delete_by_name과 delete_by_date에서 해당 연산을 사용하나 데이터를 복사하는 차이는 있으나 일단 탐색을 하고 나서는 삭제하는 연산은 같기 때문에 하나로 보여준 순서도이다. Namebst 역시 termsbst와 같은 BST의 구조를 가지기에 삭제 방법은 동일하다.

3. Algorithm

main.cpp
Main.cpp에는 Manager class 객체를 생성하고 run을 실행한다. Run을 실행할 때에 "command.txt"의 command가 저장되어 있는 파일의 이름을 보내준다. Run이 끝난 이후에 종료한다.
manager.cpp
Main.cpp에서 run을 이용해 manager.cpp에서 구동함. Command.txt에서 받아오는 명령어에 따라 총 8개의 동작을 수행. LOAD, ADD, QPOP, SEARCH, PRINT, DELETE, EXEC로 각각의 명령어에 따른 동작이 있음. 또한 명령어가 비정상적위와 같지 않을 경우에는 에러코드 1000을 출력. 모든 커맨드가 수행될 시에는 해당 프로그램을 종료한다.
LOAD, ADD, QPOP 함수 Load는 자료구조 내에 데이터가 남아있거나 파일이 정상적으로 열리지 않을 시 에러코드 100을 출력한다. 정상적인 동작은 data.txt의 파일이 끝날때까지 queue에 data를 push하는 연산을 수행하고 push한 data는 출력하도록한다. Add는 받은 데이터의 구성이 정상적인가를 먼저 판별한다. 정상적이지 않으면 에러코드 200을 출력한다. 정상적인 경우 성공 코드를 출력하고 queue에 해당 data를 push 후 추가한 데이터의 내용에 대해 출력한다. Qpop은 queue가 먼저 비어있는지 본다. Queue의 data를 pop하고 pop한 데이터를 termslist와 namebst에 insert한다. 후에 성공 코드를 출력한다. 이게 정상적인 명령이고 queue가 만약 비어 있으면 에러코드 300을 출력한다.
SEARCH, PRINT, DELETE, EXIT 함수 Search는 이름을 인자로 받아 해당 내용을 찾는데, 이름을 찾으면 해당 이름을 가지고 있는 data의 모든 정보를 출력한다. 만약 찾지 못하면 에러코드 400을 출력한다. Print는 이름을 찾는지, 계약종류를 본다. 이름으로 찾으면 namebst가 존재한다면 namebst가 갖고 있는 모든 data의 정보를 출력한다. Namebst가 없다면 에러코드 500을 출력한다. 만약 종류로 찾는데, termslist 자체가 없다면 역시 에러코드 500을 출력하게 된다. 만약 termslist가 있다면 해당 type에 맞게 검색을 한 후 해당 type에 있는 모든 data의 정보를 출력한다. Delete는 먼저 name에서 찾는지 terms에서 찾는지 판단한다. 만약 name에서 찾으면 그 이름을 namebst에서 찾고 해당 data를 삭제한다. 만약 terms인 경우 각 type에서 해당 기한 보다 계약 만료일자가 작다면 해당 노드들을 삭제한다. 삭제에 성공하면 성공 코드, 삭제에 실패하면 에러코드 600을 출력한다. Exit은 프로그램을 종료하는 코드로 모든 할당된 것을 할당해제하고 파일 스트림을 닫

고 종료한다.

MemberQueue.cpp

Memberqueue에서의 연산은 empty, full, push, pop, front가 있다.
Empty는 현재 head(front) == tail(rear)인지 판별한다. Front라는 함수가 이미 만들어져서 head로 바꾸게 되었다. 그에 맞추어 tail 로 수정하게 되었다. Full (tail + 1) % 101 == head 인지 판별한다. Push의 경우에는 먼저 큐가 차있는지 판단한다. 차있지 않은 경우 Tail의 위치를 옮기고 나서 큐에 data를 저장한다. 만약 차있는 경우에는 프로그램을 종료한다. Pop의 경우에는 먼저 큐가 비어있는지 판단한다. 비어있는 경우에 Head의 위치를 옮기고 나서 큐를 반환한다. 비어있지 않은 경우에는 프로그램을 종료한다.

TermsBST.cpp

insert에서 큐의 정보를 termsbstnode에 정보를 저장한다. Termsbstnode의 root가 비어있는 경우 루트에 해당 node를 삽입한다. Root가 비어있지 않은 경우에 삽입위치를 탐색한다. 위치를 찾은 경웨 해당 포인터에 노드를 삽입하고 종료한다.
Print의 경우에는 재귀함수적으로 움직인다. Root를 처음 받고 받은 포인터가 nullptr인지 판단한다. Nullptr이 아닌 경우에는 print에 해당 포인터의 왼쪽을 인자로 넘겨주면서 호출한다. 호출 후 포인터의 data를 출력한다. 후에 print에 해당 포인터의 오른쪽을 인자로 넘겨주면서 호출한다. 만약 nullptr의 경우에는 함수를 끝낸다.

Delete by date는 먼저 종료코드 "END"가 있다. 포인터에 루트를 위치를 저장한다. 포인터가 계약 만료일자에 따라 만약 해당 포인터의 만료일자가 빠르면 왼쪽으로 이동하고 아니면 탐색을 종료한다. 탐색 성공 여부는 해당 포인터가 nullptr이 아닌가 이다. 만약 nullptr의 경우 END 코드를 반환하고, nullptr이 아닌 경우에는 찾은 포인터의 BST의 삭제연산을 수행한다. 삭제한 이름을 반환하고 함수를 종료하다.
Delete by name은 파라미터로 큐 데어터에서 받은 이름과 계약 종료 일자를 저장한다. 계약 종료일자에 따라 탐색을 진행한다. 만약 종료일자가 빠르면 포인터를 왼쪽으로 늦으면 포인터를 오른쪽으로 간다. 만약 같다면 이름까지 비교해서 이름이 맞으면 해당 포인터로 탐색은 끝난다. 다른 경우에는 다시 오른쪽으로 간다. 탐색의 성공일 때에는 포인터가 nullptr이 아니다. 찾은 경우 bst 삭제 연산을 수행하고 아닌 경우에는 함수를 종료한다.

삭제할 포인터의 종류에 따라 삭제하는 방법을 달리한다.

1) 해당 포인터가 leafnode인 경우
   해당 포인터 이전 포인터가 존재하는가를 따진 후 이전 포인터가 없는 경우 root를 삭제한다. 만약 왼쪽으로 이동했다면 이전 포인터의 왼쪽은 nullptr, 오른쪽은 nullptr 로 설정한다.
2) 해당 포인터가 left child node만 가지는 경우

해당 포인터의 이전 포인터가 없는 경우 root를 왼쪽 노드로 옮긴다. 만약 왼쪽으로 이동했다면 이전 포인터의 왼쪽을, 오른쪽으로 이동했다면 이전 포인터의 오른쪽을 해당포인터의 왼쪽 포인터로 연결하고 해당 포인터를 삭제한다.

3) 해당 포인터가 right child node만 가지는 경우
해당 포인터의 이전 포인터가 없는 경우 root를 왼쪽 노드로 옮긴다. 만약 왼쪽으로 이동했다면 이전 포인터의 왼쪽을, 오른쪽으로 이동했다면 이전 포인터의 오른쪽을 해당포인터의 오른쪽 포인터로 연결하고 해당 포인터를 삭제한다.

4) 해당 포인터가 left & right 둘 다 가지는 경우
현재 포인터, 이전 포인터, 전전 포인터 세 개를 설정한다. 전전 포인터는 삭제할 포인터, 이전 포인터는 전전 포인터의 오른쪽, 현재 포인터는 전전 포인터의 왼쪽을 가지고 있다. 현재 포인터가 null이 될 때까지 왼쪽으로 계속 이동한다. 이전 포인터의 내용을 삭제할 포인터에 복사한 후 이전 포인터를 삭제한다. 현재 포인터가 null이 된 경우 전전 포인터가 삭제할 포인터인 경우 이동을 하지 않은 것으로 전전 포인터의 오른쪽을, 이동한 경우에는 전전 포인터의 왼쪽을 이전 포인터의 오른쪽과 연결한다.

TermsList.cpp

Insert는 다음과 같이 수행한다.
만약 head == nullptr인지 판단한다. 만약 참이면, head가 없다는 뜻으로 추가하는 노드가 head가 되는 것이다. 해당 termslistnode의 termsbstnode에 추가하는 data의 정보를 삽입한다. 해당 termslistnode의 계약종류를 설정한다.
거짓인 경우에는 termslistnode의 계약종류와 같은지를 탐색한다. 만약 다르면 다음 node로 이동하고 찾으면 해당 termsbst에 data의 정보를 삽입한다. Node를 가리키는 ptr이 nullptr인 경웨는 해당 종류를 만족하는 termslistnode가 존재하지 않는 것으로 해당 종류의 node를 생성하고 node의 termsbst에 data를 삽입, 그리고 termslist에 연결한다.
Search의 경우에는 파라미터로 받은 종류와 일치한 termslistnode의 termsbst를 반환한다. 찾지 못하면 nullptr이, 찾으면 주소가 제대로 반환되게 되는 것이다.
Delete_all의 경우에는 모든 termslistnode를 삭제하는 것이다. Head가 nullptr이 될 때까지 삭제연산을 수행한다.

Delete_by_name의 경우에는 namebst에서 삭제하고 termslist를 동기화하는 함수이다. Namebst에서 삭제한 후 얻으 큐 정보로 type을 먼저 판별한다. 해당 type을 찾고 나서는 삭제연산을 시작한다. 삭제연산이 되고 나서는 멤버수를 1 감소한다. 후에 멤버

수가 0이라면 해당 termslistnode를 삭제한다.

Delete_by_date의 경우에는 termslist에 있는 모든 node의 bst에 해당 계약만료일자보다 적은 삭제한다. 삭제하는 경우 해당 bstnode의 이름을 반환하여 namebst에 넘겨 namebst 내에서 그 이름을 가지고 bst삭제연산을 수행한다. 이 역시 "END" 코드르 이용하여 찾지 못하면 END가 변경되지 않기 때문에 이는 못찾았음으로 간주하고 다음 리스트의 노드로 이동 삭제를 반복한다. 모든 노드에서 삭제를 수행하고 나서는 멤버 수가 0임을 판별하며 0인 것은 노드를 삭제하는 연산을 수행하며 이 역시 모든 노드를 탐색하며 삭제를 수행한다.

Namebst.cpp

Namebst에서의 Insert는 큐의 정보를 우선 namebstnode에 저장한다. 이때 namebst에서의 root가 있는지 판별한다. 없으면 해당 노드를 루트에 삽입하고 종료한다. 없는 경우, 탐색할 위치를 찾는다. 삽입은 이름의 사전순을 따르며 사전순으로 앞이면 포인터가 왼쪽으로, 뒤거나 동일하면 포인터 오른쪽으로 이동하는 것을 삽입할 위치를 찾을 때까지 반복한다.

Search는 먼저 포인터가 루트를 갖는다. 이때 루트가 없는 경우 바로 탐색을 종료한다. 루트가 있는 경우에는 이름을 탐색하는 것을 시작한다. 이 때 탐색은 삽입과 마찬가지로 탐색하는 이름이 사전순으로 아이면 포인터가 왼쪽으로, 뒤이면 포인터를 오른쪽으로 움직이며 반복한다. 사전순으로 같은 것을 찾을 경우에는 해당 포인터를 반환하며 끝낸다.

Print는 재귀함수이다. 해당 함수는 처음 파라미터로 root가 주어지며 받은 포인터가 nullptr이면 출력을 종료한다. 만약 nullptr이 아닌 경우에는 다시 재귀함수로써 왼쪽 포인터를 인자로 넘겨주며 해당 함수를 호출한다. 그 후, 포인터의 데이터를 출력하고나서는 오른쪽 포인터를 인자로 넘겨주어 해당 함수를 호출한다.

Namebst에서의 delete_by_name은 이름을 인자로 받아서 삭제연산을 수행할 때 호출한다. 포인터가 루트를 먼저 받는데, 루트가 nullptr이면 nullptr을 반환한다. Nullptr이 아니라면 이름을 사전순으로 검색한다. 해당 이름보다 사전순으로 앞에 있으면 왼쪽으로 움직이고, 사전순으로 뒤에 있으면 오른쪽으로 움직인다. 이름 탐색에 성공한 경우 memberqueuenode에 해당 data를 복사한다. 찾은 포인터에 대해 BST 삭제연산을 수행하고 종료한다.

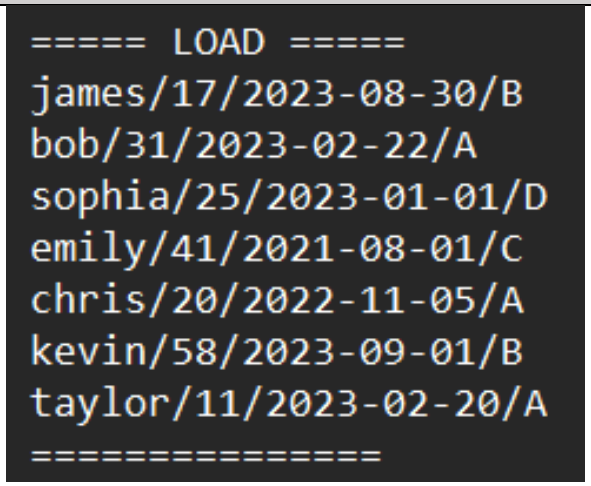
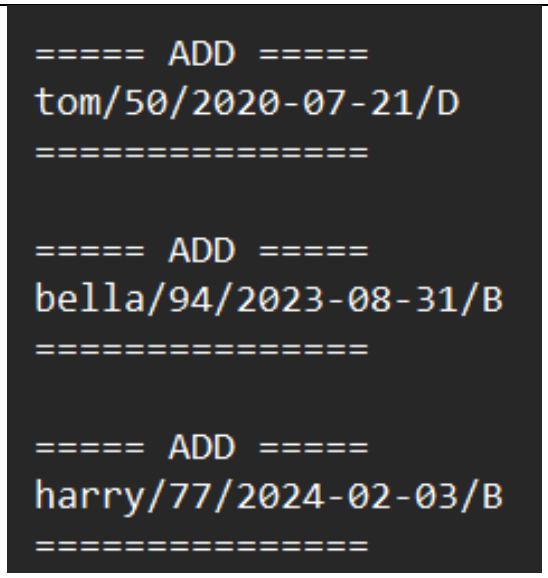
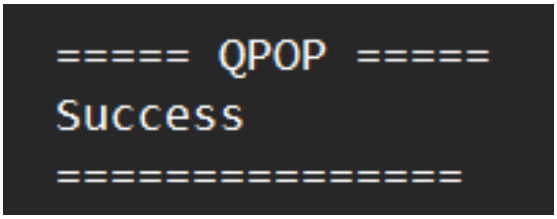
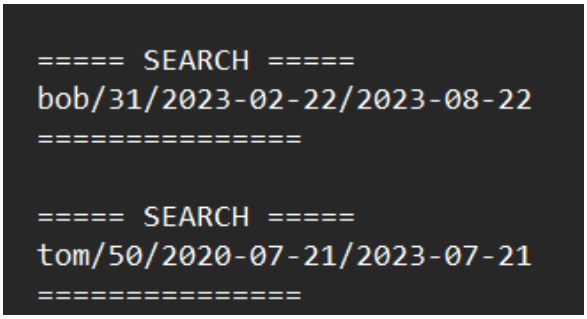
Namebst에서의 delete_by_date는 termslist에서 수행하는 delete_by_date에 의해 이름을 받아서 수행하게 된다. 이 역시 delete_by_name처럼 이름의 사전순을 이용하여 삭제를 수행한다. 앞과 다른 점은 데이터를 복사하지 않는다. 탐색에 성공하면 bst삭제연산을 수행하고 해당 함수를 종료한다.

삭제할 포인터의 종류에 따라 삭제하는 방법을 달리한다.

1) 해당 포인터가 leafnode인 경우
   해당 포인터 이전 포인터가 존재하는가를 따진 후 이전 포인터가 없는 경우 root를 삭제한다. 만약 왼쪽으로 이동했다면 이전 포인터의 왼쪽은 nullptr, 오른쪽은 nullptr 로 설정한다.
2) 해당 포인터가 left child node만 가지는 경우
   해당 포인터의 이전 포인터가 없는 경우 root를 왼쪽 노드로 옮긴다. 만약 왼쪽으로 이동했다면 이전 포인터의 왼쪽을, 오른쪽으로 이동했다면 이전 포인터의 오른쪽을 해당포인터의 왼쪽 포인터로 연결하고 해당 포인터를 삭제한다.
3) 해당 포인터가 right child node만 가지는 경우
   해당 포인터의 이전 포인터가 없는 경우 root를 왼쪽 노드로 옮긴다. 만약 왼쪽으로 이동했다면 이전 포인터의 왼쪽을, 오른쪽으로 이동했다면 이전 포인터의 오른쪽을 해당포인터의 오른쪽 포인터로 연결하고 해당 포인터를 삭제한다.
4) 해당 포인터가 left & right 둘 다 가지는 경우
   현재 포인터, 이전 포인터, 전전 포인터 세 개를 설정한다. 전전 포인터는 삭제할 포인터, 이전 포인터는 전전 포인터의 오른쪽, 현재 포인터는 전전 포인터의 왼쪽을 가지고 있다. 현재 포인터가 null이 될 때까지 왼쪽으로 계속 이동한다. 이전 포인터의 내용을 삭제할 포인터에 복사한 후 이전 포인터를 삭제한다. 현재 포인터가 null이 된 경우 전전 포인터가 삭제할 포인터인 경우 이동을 하지 않은 것으로 전전 포인터의 오른쪽을, 이동한 경우에는 전전 포인터의 왼쪽을 이전 포인터의 오른쪽과 연결한다.

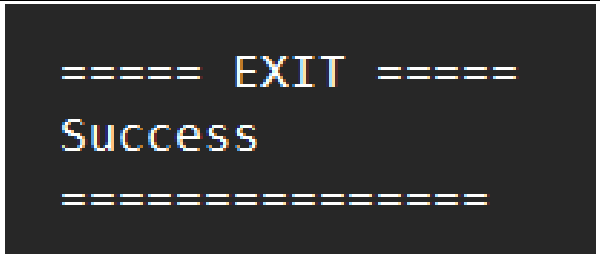
앞서 보았던 flow chart에서 언급했던 것처럼 Namebst는 Termsbst와 같이 BST의 자료구조를 가지기에 삭제 방법이 동일하다.

4. Result Screen

결과 화면	동작 설명
	command.txt 파일에 있는 LOAD 명령어에 의해 실행된다. 해당 데이터는 data.txt 파일 내에 있는 내용으로 해당 데이터 들은 LOAD 명령어를 통해 quque에 push 하여 저장한다. 저장된 데이터들을 출력한다.
	해당 결과 화면은 ADD 명령어에 의해 실행된다. 데이터를 추가할 때 'ADD 회원이름 나이 개인정보수집일자 가입약관종류" 형식으로 얻게 되며 이를 queue에 push하며 저장한 내용들을 출력한다.
	해당 내용은 QPOP 명령어에 의해 실행된다. queue에 있는 모든 데이터들이 pop되면서 namebst와 termslist로 저장되었을 때 "Success" 메시지를 출력한다.
	해당 결과 화면은 SEARCH명령어를 통해 실행된다. 이름을 이용하여 search한 결과를 출력한다. 이는 namebst에서 O(log n)의 탐색을 통해 찾은 경우 그 회원의 "회원이름/나이/개인정보수집일자/개인정보만료일자"의 형식으로 출력한다.

`` ==== PRINT ===== Name_BST bella/94/2023-08-31/2024-08-31 bob/31/2023-02-22/2023-08-22 chris/20/2022-11-05/2023-05-05 emily/41/2021-08-01/2023-08-01 harry/77/2024-02-03/2025-02-03 james/17/2023-08-30/2024-08-30 kevin/58/2023-09-01/2024-09-01 sophia/25/2023-01-01/2026-01-01 taylor/11/2023-02-20/2023-08-20 tom/50/2020-07-21/2023-07-21 =============== ```	해당 결과 화면은 PRINT라는 명령어에 의해 실행된다. 해당 내용을 출력하되 namebst에서 출력을 하게 된다. 이는 중위순회 방식으로 출력하게 되는데, 이를 이용하여 이름을 오름차순으로 출력하게 한다. 모든 가지고 있던 모든 데이터를 출력한다. "회원이름/나이/개인정보수집일자/개인정보만료일자"의 형식으로 출력한다.
`` ==== PRINT ===== Terms_BST A chris/20/2022-11-05/2023-05-05 taylor/11/2023-02-20/2023-08-20 bob/31/2023-02-22/2023-08-22 =============== ==== PRINT ===== Terms_BST B james/17/2023-08-30/2024-08-30 bella/94/2023-08-31/2024-08-31 kevin/58/2023-09-01/2024-09-01 harry/77/2024-02-03/2025-02-03 =============== ==== PRINT ===== Terms_BST C emily/41/2021-08-01/2023-08-01 =============== ==== PRINT ===== Terms_BST D tom/50/2020-07-21/2023-07-21 sophia/25/2023-01-01/2026-01-01 =============== ```	해당 결과 화면 역시 PRINT라는 명령어에 의해 실행된다. 앞과 다른 점은 Terms_BST에서 출력하게 되는데, 가입약관 종류에 따라 출력한다. 해당 내용 역시 중위순회 방식으로 출력하는데, 개인정보만료일자를 기준으로 오름차순으로 출력한다.

`` ===== DELETE ===== Success =============== ==== PRINT ===== Name_BST bella/94/2023-08-31/2024-08-31 bob/31/2023-02-22/2023-08-22 chris/20/2022-11-05/2023-05-05 harry/77/2024-02-03/2025-02-03 james/17/2023-08-30/2024-08-30 kevin/58/2023-09-01/2024-09-01 sophia/25/2023-01-01/2026-01-01 taylor/11/2023-02-20/2023-08-20 tom/50/2020-07-21/2023-07-21 =============== ===== ERROR ===== 500 =============== ```	해당 결과 화면은 "DELETE NAME emily" 수행 후 Name_BST로 출력과 emily가 속해 있었던 가입약관 종류가 C인 것을 출력한 것이다. Name_BST로 출력한 것에서 emily가 빠진 것을 확인할 수 있으며, 종류가 C인 Terms_BST 내에 있는 것은 emily 뿐으로 삭제 수행 후 Terms_BST가 완전히 사라지며 에러코드 500이 출력되는 것을 확인할 수 있다.
`` ===== DELETE ===== Success =============== ==== PRINT ===== Name_BST harry/77/2024-02-03/2025-02-03 kevin/58/2023-09-01/2024-09-01 sophia/25/2023-01-01/2026-01-01 =============== ```	해당 내용은 "DELETE DATE 2024-09-01"으로 해당 만료일자를 기준보다 작은 Terms_BST_Node 들을 삭제한다. 후에 Name_BST를 출력한다. 그러면 해당 만료일자 보다 작은 Name_BST_Node들은 삭제된 것을 확인할 수 있다.
`` ===== ERROR ===== 500 =============== ==== PRINT ===== Terms_BST B kevin/58/2023-09-01/2024-09-01 harry/77/2024-02-03/2025-02-03 =============== ==== PRINT ===== Terms_BST D sophia/25/2023-01-01/2026-01-01 =============== ```	후에 Terms_List 내에서도 삭제되었는지도 확인한다. 에러코드 500은 Terms_BST 중 가입약관 종류가 A를 출력하는데 A에 있던 모든 노드들이 이전에 삭제 연산으로 사라져 해당 BST가 없어졌기 때문에 출력된다. 이후 역시 삭제 이후에 남아있는 B, D에서의 데이터를 출력한다. 앞서 보인 Name_BST에 남아있는 데이터가 동기화 되었음을 확인할 수 있다.

	EXIT은 프로그램 종료 코드로 해당 명령어에 의해 모든 동적할당 된 data들은 동적할당 해제를 하고 열려있는 모든 file을 닫고 프로그램을 종료하게 된다.

5. Consideration

초기 queue를 구현할 때에는 링크드 리스트를 고려했다. 공간 복잡도 면에서 아주 큰 배열을 만드는 것보다 낫다고 생각했기 때문이다. 하지만 시간 복잡도를 구현했을 때 할당하고 할당해제 하는 과정에서 조금 안 좋을 수도 있다고 생각해서 환형 큐로 구현하기로 했다.

Delete를 구현하면서 해당 노드가 삭제되면 처음에는 left child와 right child 역시 소멸자에서 같이 동적할당을 해제하는 방식으로 했으나, 지속적인 오류가 났다. 이를 고치기 위해 이론적으로 다시 한번 생각을 해보았다. 주소가 있을 때, 포인터는 그 주소를 가리키기만 할 뿐 그 주소를 말하는 것이 아니다. 소멸자를 통해 그 주소의 값을 동적할당 해제를 하게 된다면 다른 노드에서 그 주소를 참조하는 포인터는 동적할당이 되지 않은 곳을 접근하려는 상태가 되어버리는 것을 인지하게 되었고, 노드의 소멸자에서 left child나 right child를 동적할당 해제하려던 것을 없애 문제를 해결했다.

termslist에서 date를 가지고 Delete를 수행할 때 namebst에서 동기화 하기 위해 termslist에서 삭제를 수행하면 해당 이름을 받아 namebst에서 삭제를 하게 만들었다. 굳이 돌아가서 검색하지 않고 한 번에 조건에 부합하는 노드들을 삭제하고 싶었으나, 해당 방법이 생각이 나지 않아 반복을 많이 돌게 만든 것이 아쉽다.

프로젝트를 완성하고 memory leak이 있는지 검사했다. 초기 검사에서는 2800byte의 memory leak이 발생했다. Delete 연산을 하면서 삭제되지 않은 node들이 존재함을 디버깅을 통해 알게 되었고, 노드 들을 다 삭제했을 때는 반환 값으로 인한 값들의 memory leak으로 400byte가 남아있었다. 해당 값들의 할당 해제로 memory leak을 완전히 없앴다. 아래는 리눅스에서 memory leak을 검사했을 때 나오는 문구이다.

```
HEAP SUMMARY:
    in use at exit: 0 bytes in 0 blocks
  total heap usage: 195 allocs, 195 frees, 262,120 bytes allocated

All heap blocks were freed -- no leaks are possible
```

예시로 준 파일들 data.txt, command.txt를 돌렸을 때 195개의 동적할당과 195개의 동적할당 해제, 총 262,120 byte가 동적할당 되었다가 해제되었다. Memory leak을 검사하면서 최대한 leak이 발생하지 않도록 코드를 짰다고 생각했으나 한번 검사해보니 메모리를 잘 관리하지 못했다는 것을 깨닫았다. 앞으로는 더 철저히 메모리 관리로 leak이 발생하지 않도록 노력하겠다.